Міністерство освіти і науки України
«Київський національний університет імені Тараса Шевченка»

Факультет інформаційних технологій
Кафедра кібербезпеки та захисту інформації

ДОПУСТИТИ ДО ЗАХИСТУ:
завідуюча кафедри кібербезпеки
та захисту інформації
_____Наталія ЛУКОВА-ЧУЙКО
«14» червня 2022р.

ПОЯСНЮВАЛЬНА ЗАПИСКА

дипломної роботи
бакалавра
(назва освітнього ступеня)

галузь знань 12 Інформаційні технології
(шифр і назва галузі знань)

спеціальність 125 Кібербезпека
(код і назва спеціальності)

освітня програма Кібербезпека
(назва освітньої програми)

на тему: «Програмна імплементація методів захисту сучасних вебдодатків від поширених загроз»

Виконавець: студент IV курсу, групи КБ-42

_____ **Котов Максим Сергійович**
(підпис) (прізвище ім'я по-батькові)

	Прізвище, ініціали	**Підпис**
Керівник	Сергій ДАКОВ.	

Нормоконтроль	Юрій ЩЕБЛАНІН.	

Київ 2022

Міністерство освіти і науки України
Київський національний університет імені Тараса Шевченка

Факультет інформаційних технологій
Кафедра кібербезпеки та захисту інформації

ЗАТВЕРДЖЕНО:
завідуюча кафедри кібербезпеки
та захисту інформації
_____Наталія ЛУКОВА-ЧУЙКО
«01» листопада 2021 р.

ЗАВДАННЯ
на виконання дипломної роботи

спеціальності 125 Кібербезпека
(код і назва спеціальності)

освітньої програми Кібербезпека
(назва освітньої програми)

Студентові **КБ-42** **Котову Максиму Сергійовичу**
(група) (прізвище ім'я по-батькові)

Тема дипломної роботи Програмна імплементація методів захисту сучасних вебдодатків від поширених загроз

1. ПІДСТАВИ ДЛЯ ПРОВЕДЕННЯ РОБОТИ

Тема дипломної роботи затверджена на засіданні кафедри кібербезпеки та захисту інформації протокол №5 від 29.10.2021 р.

2. ВИХІДНІ ДАНІ ДЛЯ ПРОВЕДЕННЯ РОБІТ

Архітектури розробки та взаємодії вебдодатків, стек технологій функціонування вебдодатків, механізми аутентифікації, алгоритми шифрування

3. ЗМІСТ РОЗРАХУНКОВО-ПОЯСНЮВАЛЬНОЇ ЗАПИСКИ

Дослідження архітектури розробки та взаємодії вебдодатків, сучасні протоколи прикладного рівня, порушення контролю доступу, ін'єкції коду, порушення криатографічного захисту, міжсайтова підробка запитів, розробка механізмів аутентифікації, застосування механізмів захисту від CSRF

4. ВИМОГИ ДО РЕЗУЛЬТАТІВ ВИКОНАННЯ РОБОТИ

Практична цінність Програмна імплементація комбінацій механізмів захисту вебдодатків та формування рекомендацій щодо їх застосування.

5. ДАТА ВИДАЧІ ЗАВДАННЯ

Дата видачі завдання: 29 жовтня 2021 року

Завдання видав _____ Сергій ДАКОВ
(підпис) (ініціали, прізвище)

Завдання прийняв до виконання _____ Максим КОТОВ
(підпис) (ініціали, прізвище)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Найменування етапів робіт	Строки виконання робіт (початок-кінець)	Відмітка про виконання
1	Уточнення постановки задачі	29.10.2021 – 10.02.2022	*виконано*
2	Аналіз літератури	11.02.2022 – 15.03.2022	*виконано*
3	Дослідження структури вебдодатків	15.03.2022 – 23.03.2022	*виконано*
4	Аналіз основних вразливостей	23.03.2022 – 14.04.2022	*виконано*
5	Визначення стеку технологій	14.04.2022 – 24.04.2022	*виконано*
6	Впровадження механізму аутентифікації на основі сесії	24.04.2022 – 08.05.2022	*виконано*
7	Впровадження механізму аутентифікації на основі токенів	09.05.2022 – 14.05.2022	*виконано*
8	Впровадження механізмів захисту від поширених вразливостей.	15.05.2022 – 20.05.2022	*виконано*
9	Формування рекомендацій щодо механізмів захисту для вебдодатків	21.05.2022 – 01.06.2022	*виконано*
10	Оформлення пояснювальної записки	02.06.2022 – 06.06.2022	*виконано*
11	Підготовка до захисту	07.06.2022 – 10.06.2022	*виконано*

Завдання видав _____ Сергій ДАКОВ
(підпис) (ініціали, прізвище)

Завдання прийняв до виконання _____ Максим КОТОВ
(підпис) (ініціали, прізвище)

Термін подання дипломної роботи до ЕК 06 червня 2022 року

РЕФЕРАТ

Пояснювальна записка дипломної роботи складається зі вступу, трьох розділів, загальних висновків, списку використаних джерел та додатків. Основний текст займає 66 сторінки, включає в себе зміст, вступ, три розділи дипломної роботи, висновки та список джерел. Крім того, робота містить 3 додатки із загальною кількістю сторінок 17. У пояснювальній записці дипломної роботи міститься 23 рисунків. Використано 37 джерел.

Метою роботи є імплементація комбінацій механізмів захисту вебдодатків від поширених загроз.

Для досягнення зазначеної мети поставлено наступні завдання:

- дослідження технологій створення вебдодатків
- проведення аналізу найбільш поширених загроз сучасних вебдодатків
- створити вебдодаток із застосуванням комбінацій механізмів захисту від найбільш поширених загроз

Об'єктом дослідження є процес виявлення загроз сучасним вебдодаткам та застосування механізмів захисту від них.

Предметом дослідження є комбінації комплексів механізмів захисту вебдодатків.

Методи дослідження: спостереження, порівняння, узагальнення, абстрагування, формалізація, аналіз і синтез.

В роботі проведено аналіз існуючих загроз безпеки сучасних вебдодатків та комбінацій механізмів захисту від них.

Запропоновано список рекомендацій щодо розробки безпечних вебдодатків.

Побудовано три вебдодатки на платформі Node.js з використанням фреймворку Express.js та різних механізмів захисту.

Розроблено список рекомендацій використання комбінацій механізмів захисту від поширених загроз.

Практичною цінністю отриманих результатів є програмна імплементація комбінацій механізмів захисту вебдодатків та формування рекомендацій щодо їх застосування.

Напрямки подальших досліджень включають криптографію, розробку безпечних серверних частин з використанням Java та Spring, вдосконалення методів захисту вебдодатків Node.js.

Ключові слова: вразливості, вебдодаток, база даних, аутентифікація на основі сесії, аутентифікація на основі токенів, ін’єкції, привілеї.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ

API	–	Application Programming Interface
IPS	–	Intrusion Prevention System
IT	–	Information Technology
SSL	–	Secure Sockets Layer
TLS	–	Transport Layer Security
VPN	–	Virtual Private Network
REST	–	Representational State Transfer
CSRF	–	Cross-Site Request Forgery
JWT	–	JSON Web Tokens
JSON	–	JavaScript Object Notation
OWASP	–	The Open Web Application Security Project
EDR	–	Endpoint Detection and Response
HTTP	–	Hypertext Transfer Protocol
EJS	–	Embedded JavaScript templates
HTML	–	HyperText Markup Language
URL	–	Uniform Resource Locator
TCP	–	Transmission Control Protocol
NPM	-	Node Package Manager

ЗМІСТ

ВСТУП

В даній дипломній роботі розглядаються основні причини суттєвої важливості безпеки вебдодатків, а також - список основних ризиків для сучасних вебдодатків. Причини, чому безпека додатків важлива, є хорошим нагадуванням про те, навіщо організаціям необхідно приділяти більше уваги безпеці вебдодатків на всіх етапах їх розробки, тестування та аж до розгортання для комерційної експлуатації.

Основні причини, чому безпека вебдодатків настільки важлива, включають:

1) запобігання втрати конфіденційних даних;
2) розуміння того, що безпека - це більше, ніж просто тестування;
3) безпека необхідна для підтримки ділової репутації та мінімізації втрат.

В наступних розділах диплому описується про найбільш поширені ризики безпеки вебдодатків відповідно до OWASP та інших джерел, що містять відомості про вразливості платформи Node.js, та зауважує, що організації повинні забезпечити захист вебпрограм від цих ризиків. Наявність безпеки мережі та системи безпеки, наприклад, брандмауери та EDR - недостатньо для захисту програми від розглянутих в дипломі загроз.

Зі збільшенням вразливостей і збільшенням кібератак, а також розширеним характером цих загроз, організаціям може знадобитися переоцінити свій підхід до захисту програм, які, ймовірно, мають вразливості, що можуть бути експлуатовані. Хоча багато організацій уже мають системну та мережеву безпеку, важливо пам’ятати про структуру безпеки, яка пропонує поглиблену архітектуру та охоплює описані загрози.

Для зараження вебсайтів, збору даних і навіть захоплення ресурсів комп’ютера використовується різноманітне шкідливе програмне забезпечення. Кількість зламаних сайтів стрімко зростає. Потенційна загроза полягає в тому, що зламані вебсайти в основному використовуються для націлювання на потенційних клієнтів, що напряму шкодить бізнесу.

РОЗДІЛ 1
АРХІТЕКТУРА ТА СПОСОБИ ВЗАЄМОДІЇ ВЕБДОДАТКІВ

1.1 Архітектура розробки вебдодатків

Архітектура розробки вебдодатків – це сукупність правил, методів та принципів за якими складаються різні компоненти, модулі та частини сучасних вебдодатків. Вона визначає, яким способом, в процесі побудови додатку, розробники інтегрують сервіси в єдину програму, здатну виконувати потреби бізнесу.

В даній роботі розглядаються три основні типи архітектури розробки вебдодатків:

1. Монолітна архітектура.
2. Мікросервісна архітектура.
3. Безсерверна архітектура.

Надійність, продуктивність і безпека продукту визначаються вебархітектурою. Команда встановлює більш точні технічні потреби для майбутнього програмного забезпечення та його рівнів і планує додаткову роботу на основі рішень, пов’язаних з високорівневою архітектурою програми (наприклад, чи буде вона монолітною чи заснованою на мікросервісах).

Для опису якості архітектури вебдодатків можна використовувати такі параметри:

1) рівень безпеки та стабільності;
2) час відповіді запиту;
3) можливість повторного використання компонентів;
4) можливість самостійно збирати аналітику та тестувати різні компоненти;
5) масштабування продукту та його компонентів.

1.1.1 Монолітна архітектура побудови

Монолітна архітектура вважається найбільш поширеним, звичайним методом розробки додатків. Монолітна архітектура вебдодатку складається з одного неподільного блоку. До такої системи зазвичай входять інтерфейс користувача на стороні клієнта, програма на стороні сервера та база даних. Вони є уніфікованими, усі функції обробляються та обслуговуються з одного місця [1, 2].

Монолітні програми зазвичай мають єдину величезну кодову базу і не мають модульності. Розробники використовують ту саму кодову базу, коли хочуть щось оновити або замінити. В результаті вони вносять зміни до всього стека одночасно.

Структура монолітної архітектури зображена на рисунку 1.1:

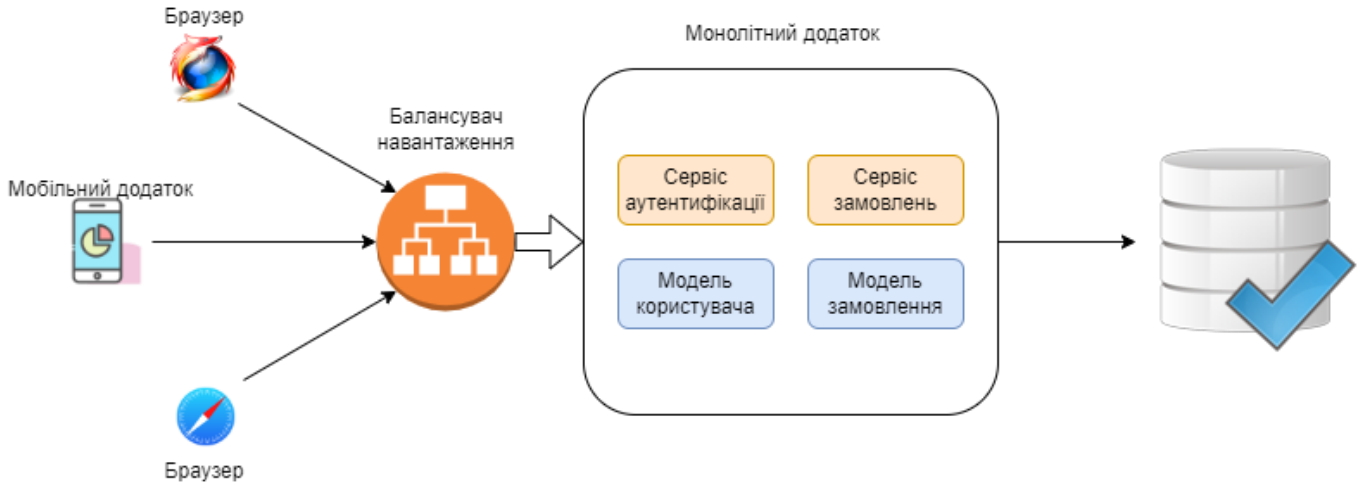

Рисунок 1.1 Структура монолітної архітектури

Створення додатку бере початок на стороні сервера, використовуючи модульну гексагональну або багатошарову конструкцію, яка складається з різних типів компонентів [5, 6]:

1. Презентація відповідає за обробку HTTP-запитів і повернення відповідей.

2. Бізнес-логіка програми відома як бізнес-логіка.

3. Об'єкти доступу до бази даних відповідають за отримання доступу до бази даних.

4. Інтеграція програми з іншими службами.

Моноліт відноситься до окремого стилю та підходу створення ведбодатків. Монолітний додаток - це однорівнева програма, яка об’єднує декілька компонентів в одну програму, яка працює на одній платформі [2, 3].

Сильні сторони монолітної архітектури включають:

1. Простий у розробці. Використання монолітної архітектури на початку проекту значно легше.

2. Тестування просте. Наприклад, можливо виконати наскрізне тестування, просто запустивши програму та використавши набір інструментів для тестування інтерфейсу користувача.

3. Легко налаштувати. Запакований додаток необхідно скопіювати на сервер і на цьому етап налаштування деплою завершується.

4. Запуск численних копій за балансиром навантаження полегшує горизонтальне масштабування.

Слабкі сторони монолітної архітектури включають:

1. Додаток занадто великий і складний, щоб належним чином зрозуміти та внести зміни вчасно та точно.

2. Розмір програми може вплинути на швидкість її запуску.

3. Кожне оновлення вимагає перерозгортання всієї програми.

4. Оскільки вплив змін рідко повністю розуміється, потрібне значне ручне тестування.

5. Коли різні модулі мають конкуруючі вимоги до ресурсів, монолітні програми може бути важко масштабувати.

6. Ще одна проблема з монолітними додатками - це відсутність надійності. Помилка в будь-якому модулі може зруйнувати всю операцію. Крім того, оскільки всі екземпляри програми ідентичні, помилка вплине на доступність всієї програми.

1.1.2 Мікросервісна архітектура побудови

Архітектура мікросервісів є методом або стилем розробки додатків у розподіленому вигляді. Це тягне за собою розбиття великих програм на менші функціональні блоки, які можуть працювати та взаємодіяти незалежно [4, 5].

Архітектура мікросервісів була створена для вирішення проблем і труднощів, які представляють монолітні архітектурні підходи до розробки додатків.

Монолітна архітектура - це величезний контейнер, який містить усі програмні компоненти програми: інтерфейс користувача, бізнес-рівень та інтерфейс даних. Це має різні недоліки, включаючи негнучкість, недостатню надійність, труднощі масштабування та затримку розвитку. Архітектура мікросервісів мала на меті уникнути цих проблем.

Структура мікросервісної архітектури зображена на рисунку 1.2:

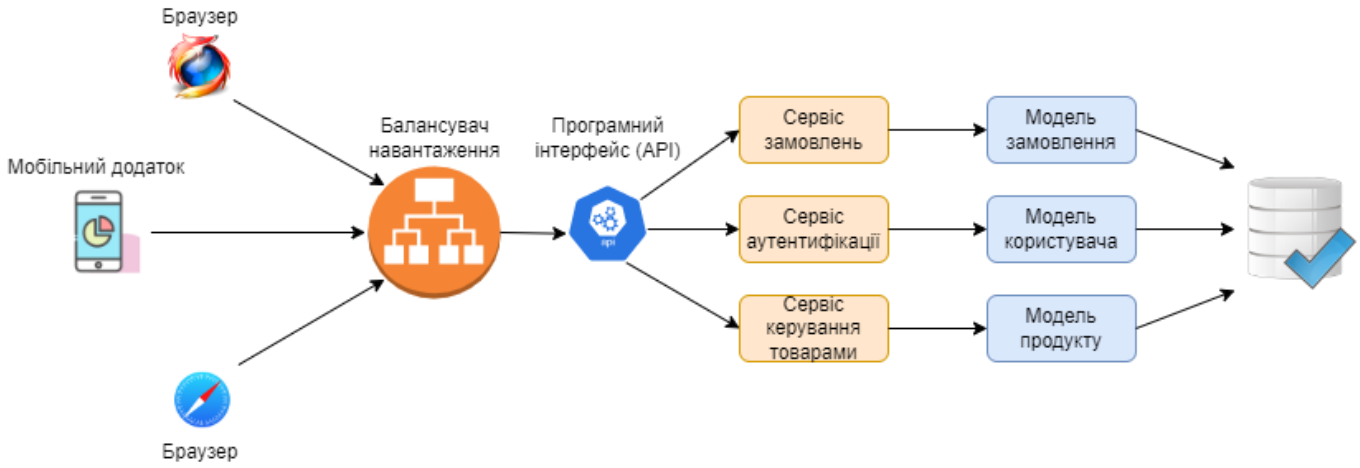

Рисунок 1.2 Структура мікросервісної архітектури

Сильні сторони мікросервісної архітектури включають:

1. Дизайн мікросервісів дозволяє розробникам створювати та розгортати служби окремо.

2. Мікросервіс може бути створений невеликою групою людей.

3. Для написання коду для різних служб можна використовувати різні мови (хоча багато практиків відмовляються від цього).

4. Для розробників, код легко зрозуміти та змінити, що може допомогти новому члену команди швидко освоюватися.

5. Оскільки вебконтейнер запускається швидше, розгортання відбувається також швидше.

6. Лише відповідну службу можна змінювати та повторно розгортати, коли потрібні зміни в певному розділі програми - не потрібно змінювати та повторно розгортати всю програму.

7. Краща ізоляція несправностей: якщо один мікросервіс виходить з ладу, інші продовжуватимуть працювати (хоча одна проблемна область монолітного додатка може поставити під загрозу всю систему).

Слабкі сторони мікросервісної архітектури включають:

1. Тестування може стати важким і тривалим через широке розгортання.

2. Інформаційні перешкоди можуть виникнути при зростанні кількості послуг.

3. Архітектура додає складності, оскільки розробники повинні враховувати відмовостійкість, затримку мережі, діапазон форматів повідомлень і балансування навантаження.

4. Оскільки це розподілена система, може виникнути дублювання роботи.

5. Обробляти випадки використання, які охоплюють багато служб без використання розподілених транзакцій, не тільки складно, але також вимагає співпраці та спілкування між різними командами.

1.1.3 Безсерверна архітектура побудови

Безсерверна архітектура - це парадигма проектування програмного забезпечення, яка дозволяє розробникам створювати та виконувати служби, не турбуючись про інфраструктуру. Розробники можуть писати та публікувати код, а хмарні провайдери налаштовують сервери для виконання своїх програм, баз даних і систем зберігання даних будь-якого розміру [7, 8].

Незважаючи на той факт, що безсерверна архітектура усуває необхідність обслуговування сервера, все ще існує висока крива навчання, особливо при об’єднанні численних функцій разом для формування складних робочих процесів у додатку.

Структура безсерверної архітектури зображена на рисунку 1.3:

Рисунок 1.3 Структура безсерверної архітектури

Команди повинні підтримувати серверне обладнання, оновлювати програмне забезпечення та систему безпеки, а також створювати резервні копії на випадок поломки. Розробники можуть передати ці завдання сторонньому постачальнику, використовуючи архітектуру без сервера, що дозволить їм зосередитися на створенні коду програми.

1.2 Архітектура взаємодії вебдодатків

У 2000 році почали з'являтися перші API. Одними з перших їх розробили Salesforce і eBay. Ці компанії бажали, щоб люди з різними потребами в даних отримували доступ до своїх даних без необхідності писати складний код або чекати кілька днів, поки чиясь інша система відповість. Розробники API обрали протокол SOAP, оскільки він простий і підтримує формат XML, який також простий. Існували

попередні протоколи, такі як CORBA та RPC, які були раніше, але вони не могли досягти такого рівня, як ці нові.

Архітектура взаємодії вебдодатків – визначає яким чином та у якому форматі різні частини вебдодатку можуть обмінюватись інформацією між собою. В контексті клієнт-серверної структури сучасних вебдодатків, архітектура їх взаємодії визначає, яким чином клієнт може отримати початкові та додаткові дані від сервера за наявної потреби.

REST і GraphQL - це дві технології отримання даних. Перша є більш традиційним способом, тоді як інша була випущена в 2015 році і швидко завоювала популярність серед розробників.

В даній роботі розглядаються три основні типи архітектури розробки вебдодатків:

1. Архітектура взаємодії Representational State Transfer.
2. Архітектура взаємодії на основі генерації шаблонів.
3. Архітектура взаємодії GraphQL.

API є основою сучасної складної програмної екосистеми. API забезпечують ці інтеграції поза лаштунками, і існує майже необмежена кількість методів для підключення різних вебдодатків. Питання того, як функціонують API, варто досліджувати, якщо існує потреба пов’язувати програмне забезпечення чи сервіс із цифровим світом.

1.2.1 Архітектура взаємодії Representational State Transfer

REST - це архітектурний підхід програмного забезпечення, який базується на правилах визначення та доступу до ресурсів. REST користується значною популярністю серед розробників вебдодатків. Без REST API важко уявити нинішній стан Інтернету [9].

Реалізації клієнта і сервера в архітектурному стилі REST можна виконувати незалежно один від одного, не знаючи про іншого. Це означає, що код клієнта

можна змінити в будь-який момент без впливу на роботу сервера, а код сервера можна змінити, не впливаючи на роботу клієнта.

Їх можна зберігати модульними та незалежними, доки одна сторона знає, який тип комунікації передати іншій. Покращується гнучкість інтерфейсу на різних платформах і масштабованість, відокремлюючи проблеми інтерфейсу користувача від проблем зберігання даних, спрощуючи компоненти сервера. Крім того, поділ дозволяє кожному компоненту розвиватися незалежно [10].

Схема роботи архітектури REST зображена на рисунку 1.4:

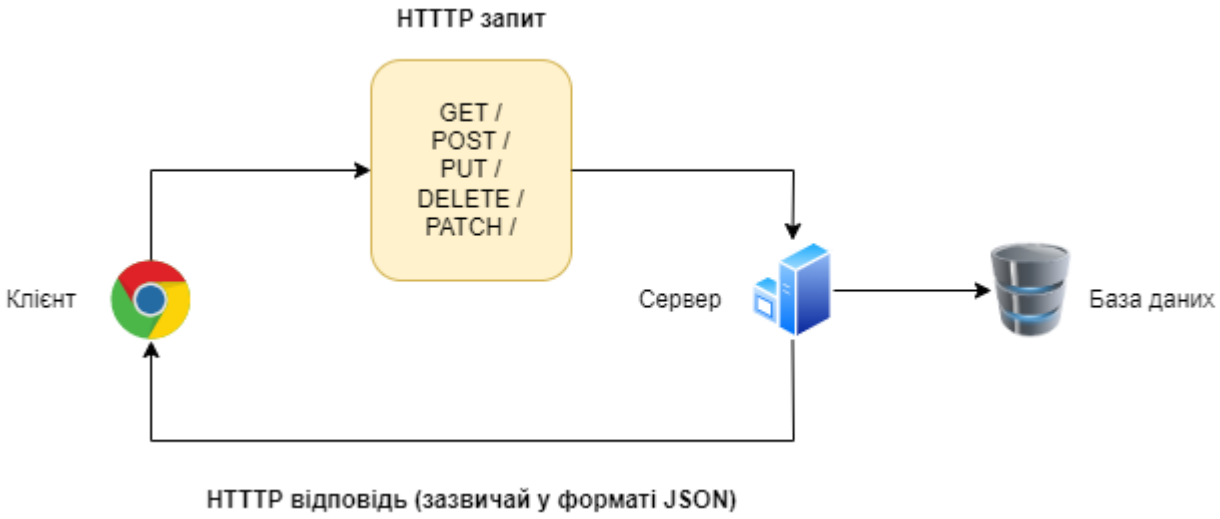

Рисунок 1.4 Схема роботи архітектури REST

У системі REST, в основному, використовуються чотири типи запитів HTTP для зв’язку з ресурсами:

1. GET - отримує один ресурс (за ідентифікатором) або групу ресурсів.
2. POST - створює новий ресурс.
3. PUT - замінює конкретний ресурс (за ідентифікатором)
4. DELETE - видаляє ресурс на основі його ідентифікатора.
5. PATCH – оновлює вказаний ресурс.

Обмеження, що диктуються архітектурою:

1. Інтерфейс завчасно узгоджений. У запиті слід вказати конкретні ресурси. Уніфіковані ідентифікатори ресурсів (URI) зазвичай використовуються для їх опису. Крім того, представлення ресурсів відрізняється від внутрішньої реалізації. Також,

представлення має містити необхідну інформацію для зміни або видалення ресурсу, а також отримання додаткових даних.

2. REST API – не має стану. Простіше кажучи, це означає, що інформація про сеанс користувача не зберігається. Таким чином, кожен запит повинен містити всю необхідну інформацію для обробки.

3. Відповідь сервера має вказати, чи потрібно кешувати його та як довго. Кешування часто оновлюваних даних покращує продуктивність і зменшує кількість зайвих обмінів клієнт-сервер.

4. В архітектурі програми потрібно кілька шарів. Між клієнтом і кінцевим сервером є кілька проміжних серверів, і кожен рівень нічого не знає про будь-який інший рівень, крім поточного. Полегшуючи балансування навантаження та забезпечуючи спільні кеші, проміжні сервери можуть підвищити доступність системи.

5. Код на вимогу - це необов’язкове обмеження, яке вказує, що потрібно надати клієнтові виконуваний код, якщо це необхідно.

1.2.2 Архітектура взаємодії на основі генерації шаблонів

Коли існує потреба швидко створити вебдодатки з кількома компонентами, у нагоді стають механізми шаблонів. Шаблони також дозволяють швидко відображати дані на стороні сервера, які необхідно надати програмі.

Механізми шаблонів зазвичай використовуються для серверних програм, які не створюються як API і працюють на одному сервері. Ejs, Jade, Pug, Mustache, HandlebarsJS є одними з найпопулярніших [11].

Коли використовується механізм шаблонів для створення серверної програми, він замінює змінні у файлі шаблону фактичними значеннями і представляє їх клієнту. Це полегшує розробку програми.

Схема роботи архітектури на основі генерації шаблонів на рисунку 1.5:

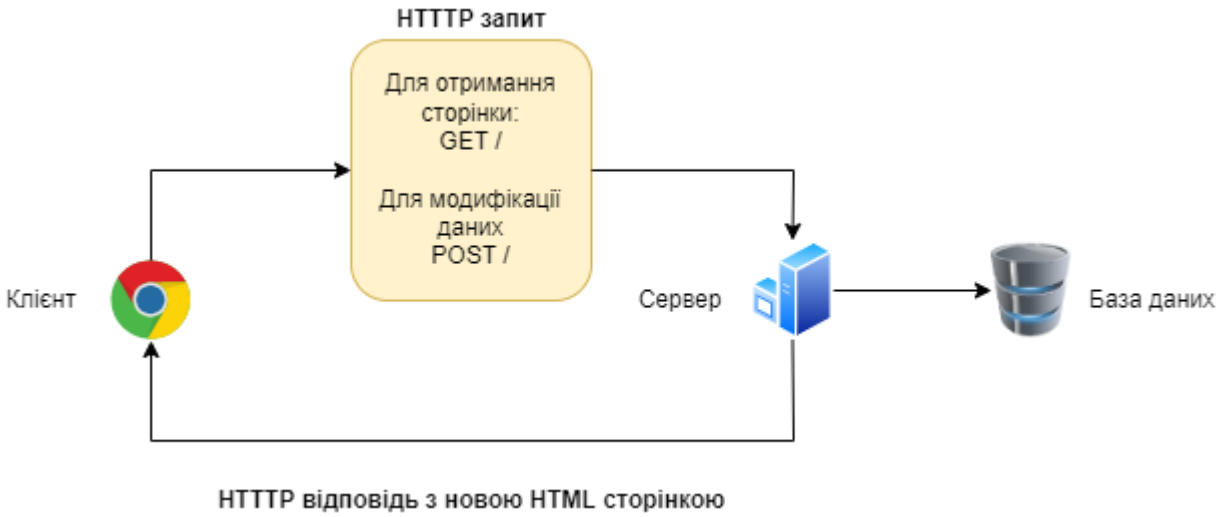

Рисунок 1.5 Схема роботи архітектури на основі генерації шаблонів

Хоча можна відтворювати статичні вебсторінки з сервера, цей метод має ряд недоліків, включаючи дублювання коду та недостатню гнучкість, особливо при отриманні даних з бази даних. Express.js містить механізм шаблонів, який дозволяє створювати динамічні сторінки HTML із серверних програм.

Механізм шаблонів працює просто: створюється шаблон і вводяться в нього змінні, використовуючи відповідний синтаксис. Потім змінним, оголошеним у файлі шаблону, призначаються значення на відповідному маршруті для відтворення шаблону. Під час візуалізації шаблону вони збираються в режимі реального часу.

При розробці вебдодатків для цієї дипломної роботи, було використано два різних генератори шаблонів: Handlebars та EJS.

Handlebars - це механізм шаблонів без логіки, який динамічно створює HTML-сторінку. Це розширення Mustache з додатковими функціями. Handlebars має мало логіки завдяки використанню кількох помічників, наприклад, if, with, unless, each та інші. Насправді, можливо вважати Handlebars суперсетом Mustache [12].

Початковий вираз допоміжного блоку матиме символ # перед ключовим словом, а кінцевий вираз матиме символ /, за яким слідуватиме те саме ключове слово, що вказує на кінець.

Handlebars.js - бібліотека Javascript, яка дозволяє створювати вебшаблони для багаторазового використання. Для створення шаблонів використовуються HTML, текст і вирази. Вирази укладені в подвійні фігурні дужки і включені в текст html.

Embedded Javascript скорочено називається EJS. Це проста мова шаблонів/система, яка дозволяє користувачам створювати HTML, використовуючи лише javascript. Коли потрібно вивести HTML із великою кількістю JavaScript, EJS стане в нагоді.

Переваги механізму шаблонів у Node.js:

1. Підвищує ефективність розробника.
2. Покращено читабельність і ремонтопридатність.
3. Покращена продуктивність.
4. Обробка на стороні клієнта оптимізована.
5. Для кількох сторінок використовуйте один шаблон.
6. CDN надає доступ до шаблонів.

1.2.3 Архітектура взаємодії на основі GraphQL

Останніми роками розробка додатків стає дедалі складнішою. Часи простого отримання даних з бази даних і представлення тексту на сторінці за допомогою форми давно минули.

Користувачі та клієнти очікують, що розробники додатків запропонують багатий, складний користувальницький інтерфейс та незабутній досвід роботи в Інтернеті, використовуючи дані з низки джерел.

На жаль, зі збільшенням складності, з класичними підходами API, такими як REST, може стати важко працювати. З цих причин було створено технологію GraphQL.

GraphQL - це новий підхід до комунікації клієнт-сервер, який спрямований на революцію в тому, як веброзробники створюють програми.

Схема роботи архітектури на основі GraphQL на рисунку 1.6:

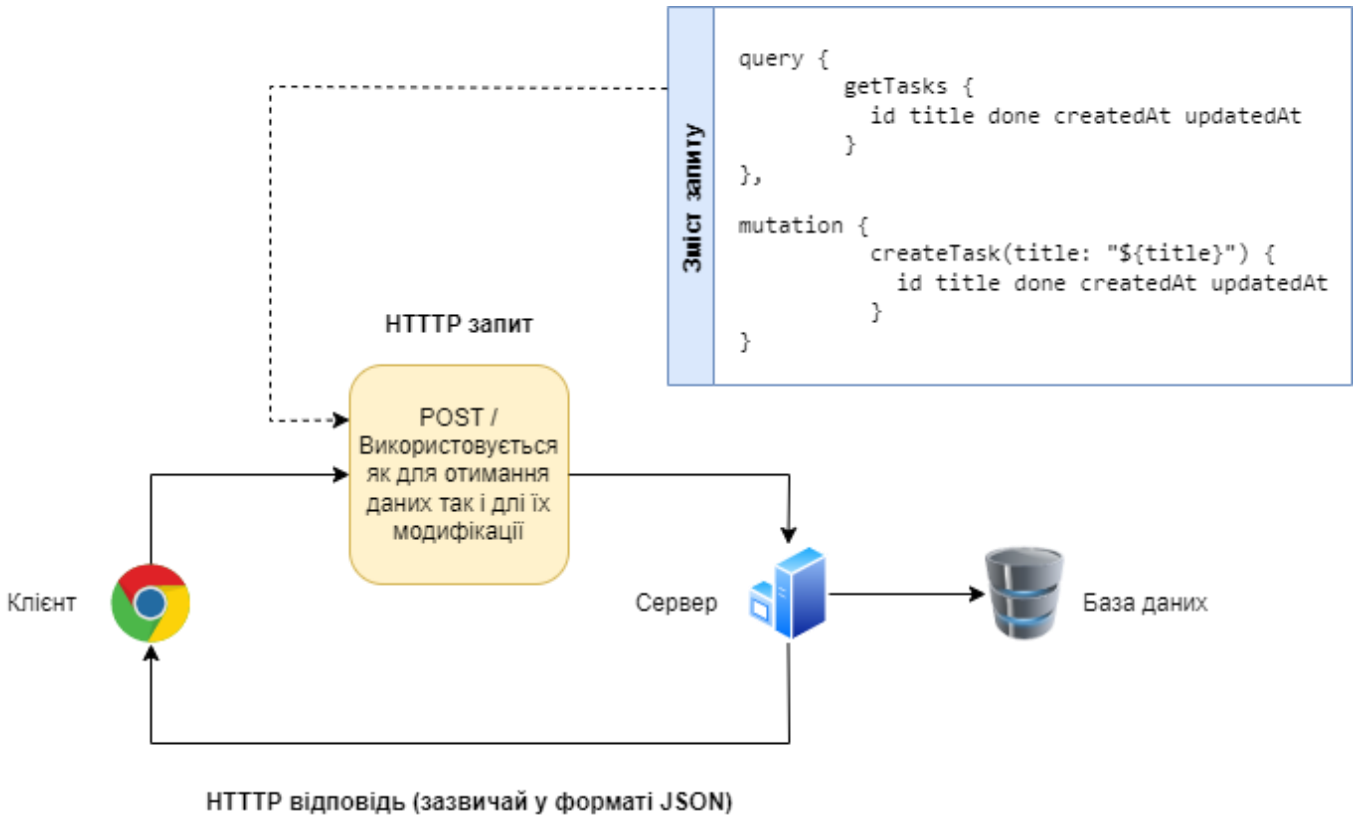

Рисунок 1.6 Схема роботи архітектури на основі GraphQL

Простіше кажучи, GraphQL - це мова запитів для API та середовище виконання для задоволення цих запитів наявними даними. GraphQL дає клієнтам можливість запитувати саме те, що їм потрібно, і нічого більше, полегшує розробку API з часом і забезпечує потужні інструменти розробника, надаючи чіткий і зрозумілий опис даних у створеному API [13].

Привабливість GraphQL здебільшого випливає з концепції просити лише те, що вам потрібно, і надати саме це - ні більше, ні менше. GraphQL пропонує досить передбачувану відповідь під час надсилання запитів до API, без надмірного надсилання даних або недобору того, що потрібно, це гарантує, що програми GraphQL є швидкими, стабільними та масштабованими.

GraphQL - це формалізований синтаксис для запиту даних із сервера, і він використовується як для завантаження даних до сервера так і передачі даних до клієнта. Існує три основні властивості GraphQL [13]:

1. Надає можливість визначення конкретного набору даних.

2. Полегшує комбінування колекцій даних.

3. Впроваджує систему типів для опису даних.

Вирішення питання вибору між REST та GraphQL залежить від програми. GraphQL долає проблеми як з надмірною, так і з недостатньою вибіркою, дозволяючи клієнту запитувати лише необхідні дані. Робота з GraphQL ефективніша, ніж робота з REST, тому розробка займає значно менше часу.

GraphQL може бути найкращим варіантом, якщо потрібно впровадити щось нове у свою розробку API, використовуючи сучасний стиль дизайну, який не вимагає багато поїздок туди й назад для отримання даних.

Однак, якщо потрібно використовувати перевірене рішення, яке включає в себе власну автентифікацію та кешування, REST – випробуваний роками стиль.

1.3 Сучасні протоколи мережевої взаємодії

Всесвітня павутина найчастіше пов’язана з протоколом передачі гіпертексту (HTTP). В Інтернеті це один із найбільш використовуваних протоколів передачі. Протокол передачі файлів (FTP) і простий протокол передачі пошти (SMTP) є двома іншими (SMTP). HTTP є найбільш широко використовуваним протоколом транспортування вебслужб, і він є основним для взаємодії REST.

HTTPS - це безпечний протокол HTTP, який дозволяє безпечно надсилати та отримувати дані через Інтернет. Щоб мати безпечний Інтернет, тепер потрібно, щоб усі вебсайти використовували протокол HTTPS. Якщо сайт не обслуговується через HTTPS, браузери, такі як Google Chrome, відображатимуть сповіщення з повідомленням «Не захищено» в рядку URL-адрес.

Для передачі даних він використовує порт 443. Шифруючи весь зв’язок за допомогою SSL, він забезпечує безпечні транзакції. Він поєднує протокол SSL/TLS з HTTP. Ідентифікує мережевий сервер зашифрованим і безпечним способом.

Транзакції SSL обговорюються за допомогою методу шифрування на основі ключа в протоколі HTTPS. Міцність цього ключа зазвичай становить 128 біт.

HTTP - це односторонній протокол, за яким клієнт надсилає запит, а сервер відповідає. Скажімо, користувач надсилає запит HTTP або HTTPS на сервер; після отримання запиту сервер надсилає відповідь клієнту; кожен запит асоціюється з відповідною відповіддю; після відправки відповіді з'єднання закривається; кожен запит HTTP або HTTPS щоразу встановлює нове з'єднання з сервером. Після отримання відповіді з'єднання розривається само собою.

HTTP - це протокол без стану, який працює поверх TCP, який є протоколом, орієнтованим на підключення. Він використовує тристороннє рукостискання для забезпечення доставки пакетів даних і повторно передає пропущені пакети.

Клієнти очікують, що інформація буде доступна миттєво в сучасному середовищі з високими зв’язками та завжди онлайн. WebSockets є одним із багатьох інструментів, доступних для розробки онлайн-додатків, які дозволяють оновлювати в режимі реального часу та спілкуватися.

Протокол WebSocket дозволяє клієнту і серверу спілкуватися в повнодуплексному двонаправленому режимі. З’єднання WebSocket унікальні тим, що вони мають двосторонній потік, що дозволяє їм швидко та ефективно надсилати дані. Хоча WebSockets має багато додатків, є деякі ситуації, коли альтернативний метод, наприклад періодичне опитування, є кращим [14].

Схема порівняння роботи протоколів прикладного рівня WebSocket та HTTP зображена на рисунку 1.5:

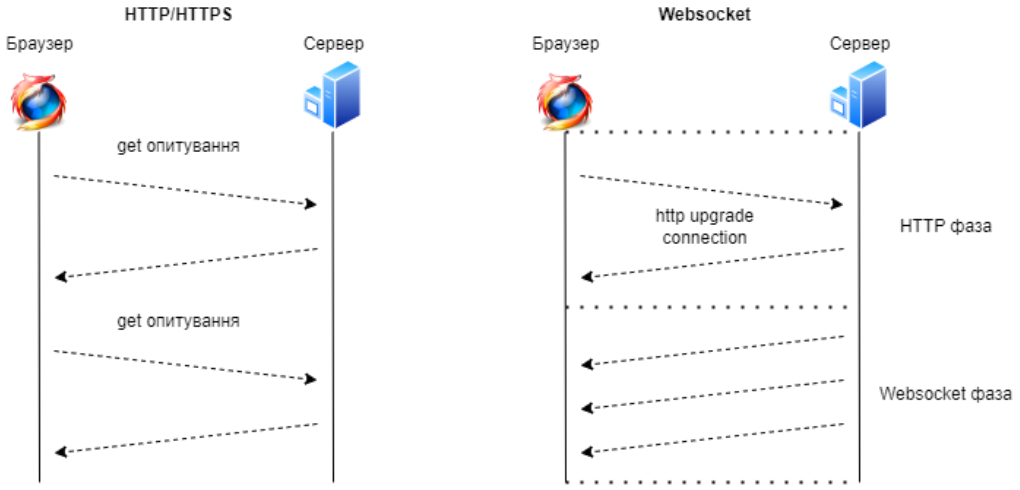

Рисунок 1.7 Схема порівняння роботи протоколу WebSocket та HTTP

WebSockets починаються як звичайні HTTP-запити та відповіді. Клієнт просить встановити з’єднання WebSocket, а сервер відповідає. Якщо початкове рукостискання пройшло успішно, клієнт і сервер погодилися використовувати існуюче з’єднання TCP/IP для запиту HTTP як з’єднання WebSocket. Для надсилання даних через це з’єднання тепер можна використовувати елементарний протокол повідомлень із рамками. З’єднання TCP припиняється після того, як усі сторони погодяться, що з’єднання WebSocket має бути припинено.

1.4 Постановка завдання

Коли справа доходить до розробки вебдодатків або будь-яких інших проектів розробки, технології з відкритим кодом стали першим вибором для спільноти розробників у всьому світі. Більшість організацій у 2021 році використовували технології з відкритим кодом для проектів розробки програмного забезпечення. Однак безпека залишається однією з головних проблем для розробників, які використовують ці технології з відкритим кодом.

Node.js є однією з таких технологій, які компанії використовують для розробки вебдодатків. Він розроблений так, щоб бути відносно безпечним. Однак під час розробки будь-якого вебдодатка за допомогою Node.js - потрібно буде використовувати різні сторонні пакети з відкритим вихідним кодом через NPM (Node Package Manager) і на жаль, згідно з опитуванням, значна частина екосистеми NPM вразливі до загроз безпеки. Зрештою, ці вразливості переносяться у вебдодатки Node.js.

Вразливості безпеки не є чимось новим для серверних фреймворків з відкритим вихідним кодом і кожен розробник Node.js розуміє ризики, які можуть становити хакери для програм і даних користувачів.

Задачею даної дипломної роботи є дослідження найбільш поширених вразливостей сучасних вебдодатків на базі Node.js, розробка таких вебдодатків з

використанням комбінацій механізмів захисту від розглянутих загроз, а також формування рекомендацій щодо розробки безпечних додатків.

Висновки за розділом 1

Архітектуру зазвичай визначають як фундаментальну організацію вебдодатка. Вона описує внутрішню структуру високорівневої системи програмного забезпечення та окреслює способи з’єднання основних її частин.

Архітектура вебдодатків являє собою набір рішень, які необхідно прийняти на початку проекту. Ці початкові рішення зазвичай мають далекосяжні наслідки для всіх аспектів процесу розробки програми.

1) розподіл фінансових ресурсів: витрати на розробку та обслуговування вебдодатків;

2) рішення щодо найму: кількість людей у команді та їхні ролі;

3) структура розбивки роботи: ієрархія завдань для виконання командою;

4) час виходу на ринок: відсутність затримок, викликаних змінами на пізніх стадіях;

5) майбутні якості продукту: продуктивність, відмовостійкість, масштабованість та надійність;

6) простота обслуговування: виправлення помилок, заміна або додавання нових функцій.

Ідеальної архітектури, яка б гарантувала, що всі аспекти веб-програми крутиться, не існує. Кожен конкретний випадок має свої вимоги та передумови. Вибір архітектури веб-програми часто вимагає прийняття правильних компромісів залежно від бізнес-цілей та пріоритетів.

РОЗДІЛ 2
ПОШИРЕНІ ВРАЗЛИВОСТІ СУЧАСНИХ ВЕБДОДАТКІВ

2.1 Порушений контроль доступу

Зловмисний користувач може отримати розширений доступ до ресурсів, які зазвичай повинні бути для нього недоступні, використовуючи помилку, недолік дизайну або проблему конфігурації в програмі чи операційній системі. Зловмисник може використовувати нещодавно отриманий доступ для крадіжки конфіденційної інформації, виконання адміністративних команд або розгортання шкідливого програмного забезпечення, завдаючи катастрофічної шкоди операційній системі, серверним додаткам, організації та репутації [15].

Зловмисники починають з виявлення слабких місць у захисті та отримання доступу до системи. У багатьох випадках початкова точка проникнення не надасть зловмисникам необхідний доступ або дані. Далі вони спробують підвищити привілеї, щоб отримати додаткові дозволи або доступ до більш чутливих систем.

За деяких обставин зловмисники, намагаючись підвищити привілеї, виявляють помилки конфігурації безпеки - недостатній контроль безпеки або недотримання принципу найменших привілеїв, що призводить до того, що користувачі мають більші привілеї, ніж їм потрібно. У деяких випадках зловмисники користуються перевагами недоліків програмного забезпечення або використовують спеціальні способи, щоб обійти схему дозволів операційної системи.

Хоча підвищення привілеїв рідко є кінцевою метою зловмисника, воно часто використовується напередодні більш цілеспрямованої кібератаки, що дозволяє зловмисникам розподіляти шкідливе корисне навантаження, змінювати параметри безпеки та відкривати нові шляхи атаки в цільовій системі.

2.1.1 Вертикальна ескалація привілеїв

Вертикальна ескалація привілеїв відбувається, коли зловмисник використовує зламаний обліковий запис для отримання додаткових прав або доступу. Наприклад, зловмисник отримує контроль над звичайним обліковим записом користувача в мережі і намагається отримати адміністративний або root-доступ. Це вимагатиме більшої витонченості і може проявлятися як розширена постійна загроза [15].

Завдяки вертикальному контролю доступу різні типи користувачів мають доступ до різних функцій програми. Адміністратор, наприклад, може мати можливість змінити або видалити обліковий запис будь-якого користувача, тоді як звичайний користувач цього не має. Вертикальний контроль доступу - це більш детальні версії моделей безпеки, які забезпечують дотримання бізнес-політики, як-от поділ ролей і найменші привілеї.

Схема вертикальної ескалації привілеїв зображена на рисунку 2.1:

Рисунок 2.1 Схема вертикальної ескалації привілеїв

Вертикальна ескалація привілеїв, також відома як атака підвищення привілеїв, тягне за собою надання користувачеві, додатку чи іншому активу більше привілеїв/привілейованого доступу, ніж вони вже мають. Потрібен перехід від низького рівня привілейованого доступу до більш високого. Вертикальна ескалація привілеїв може вимагати серії проміжних етапів (наприклад, атаки переповнення буфера), щоб обійти або скасувати контроль безпеки, використати дефекти

програмного забезпечення, мікропрограми чи ядра, або отримати привілейовані облікові дані для інших програм або самої операційної системи [15].

Фішингові електронні листи зазвичай використовуються зловмисниками, щоб отримати прямий доступ до фінансового рахунку користувача, зокрема до банківських рахунків та облікових записів електронної комерції. У листі часто вказується, що якщо споживач не натисне посилання в електронному листі та не ввійде у свій обліковий запис, його обліковий запис буде скасовано через неактивність. Це посилання переведе на вебсайт хакера, який здається максимально близьким до справжнього. Якщо користувач увійде на цей фіктивний вебсайт, зловмисник може вкрасти дані користувача та вивести кошти зі справжнього облікового запису.

Є багато недоліків, які можуть призвести до збільшення привілеїв. Міжсайтові сценарії, неправильна обробка файлів cookie та слабкі паролі є одними з найпоширеніших. Програмно можна запобігти міжсайтовим сценаріям і неналежному управлінню файлами cookie. Для усунення слабких паролів необхідні освіта кінцевого користувача та вимоги до паролів. Можливо вказати критерії складності пароля та вікові обмеження. Є два альтернативні підходи для запобігання ескалації привілеїв, які широко використовуються. Це принципи найменших привілеїв і поділу привілеїв.

Що стосується програмного забезпечення, принцип найменших привілеїв стверджує, що програмні модулі або процеси повинні мати право виконувати лише ті завдання, для яких вони створені. Жодні інші компоненти програми, операційної системи чи файлової системи не повинні бути доступні для модуля. Якщо вразливість у цьому процесі буде виявлена та використана, зловмисник матиме доступ лише до невеликої частини системи.

Поняття найменших привілеїв йде рука об руку з поділом привілеїв. Поділ програмного забезпечення або процесу на менші компоненти відомий як розділення привілеїв. Кожен із цих компонентів несе певні обов’язки.

2.1.2 Горизонтальна ескалація привілеїв

Горизонтальне підвищення привілеїв тягне за собою отримання доступу до прав іншого облікового запису з ідентичними привілеями, будь то люди чи машини. Зазвичай це тягне за собою облікові записи нижчого рівня (тобто стандартного користувача), які можуть бути вразливими до атак. Сфера доступу зловмисника з еквівалентними правами зростає з кожним новим зламаним горизонтальним обліковим записом [15].

Схема горизонтальної ескалації привілеїв зображена на рисунку 2.2:

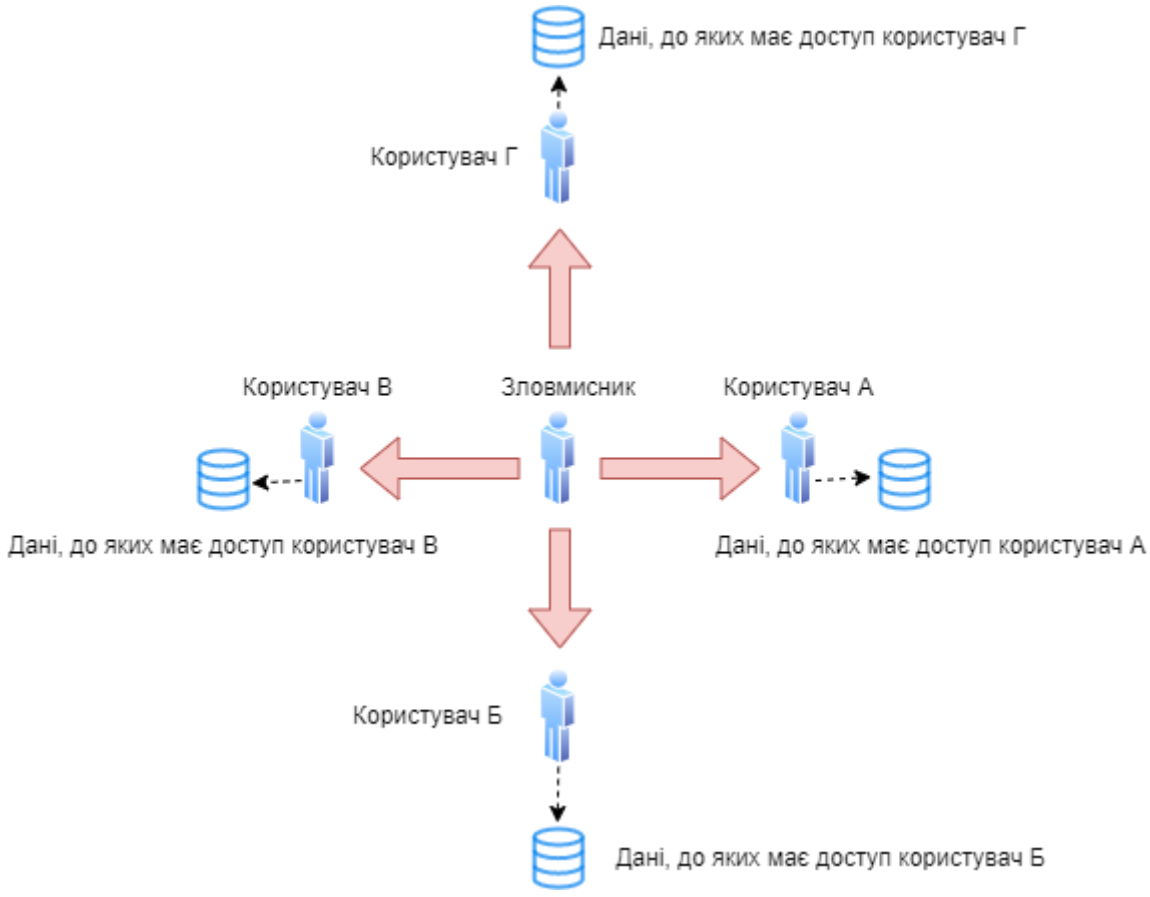

Рисунок 2.2 Схема горизонтальної ескалації привілеїв

Горизонтальне підвищення привілеїв важко здійснити, оскільки для цього потрібен як доступ до облікових даних облікового запису, так і підвищення дозволів. Ця форма нападу зазвичай вимагає глибокого знання вразливостей, які

впливають на конкретні операційні системи, а також використання інструментів злому.

Перший крок атаки для отримання доступу до облікового запису зазвичай здійснюється за допомогою фішингу. Коли справа доходить до надання підвищених прав, у зловмисника є кілька альтернатив. Щоб отримати доступ на системному або кореневому рівні, одним із способів є атака на вразливі місця в операційній системі. Наступним підходом є використання інструментів злому, таких як Metasploit, щоб допомогти з виконанням поставленої задачі.

Якщо зловмисник успішно завершив атаку горизонтальної ескалації привілеїв і тепер має доступ до перевірки та зміни запитів колег, він вирішує скористатися цим доступом для підвищення продуктивності. Зловмисник видаляє кілька запитів колег і використовує власний обліковий запис, щоб відтворити їх. Тепер зловмисник «знімав» численні облікові записи та збільшив обсяги виробництва, дозволивши йому ще на один прибутковий місяць.

Горизонтальний контроль доступу - це системи, які обмежують доступ до ресурсів для людей, яким надано на це дозвіл.

Однотипні, однорангові користувачі системи – зазвичай мають доступ до ресурсів, даних одного і того ж виду, використовуючи горизонтальні елементи керування доступом. Наприклад, банківська програма дозволить користувачеві контролювати транзакції та здійснювати платежі зі своїх рахунків, але не з рахунків інших користувачів.

2.2 Ін’єкції зловмисного коду

Зловмисна ін’єкція або введення коду в програму - відома як ін’єкція коду. Введений код може порушити цілісність бази даних, а також конфіденційність, безпеку і навіть коректність даних. Він також має можливість викрасти дані та/або обійти контроль доступу та аутентифікації. Атаки з введенням коду можуть завдати шкоди додаткам, робота яких покладається на введення даних від користувача [16].

Шкідливий код вбудовується у вихідний код, який зчитується та виконується завдяки недолікам можливості ін’єкції коду. Під час ін’єкції зловмисники користуються тим фактом, що ці системи створюють частину сегмента коду, використовуючи зовнішні дані, але не перевіряють введені дані. Шкідливий код часто пишеться для маніпулювання потоком даних, що призводить до втрати даних і зниження доступності програми.

Проблеми перевірки користувацьких даних, такі як формат даних, допустимі символи та очікуваний обсяг даних, визначаються зловмисниками та використовуються для створення шкідливого коду. Вразливості ін’єкцій займають місце серед найбільш розповсюджених загроз безпеки. Вразливості ін’єкцій дещо важко використовувати якщо при дизайні та розробці вебдодатку використано правильні практики та швидке виявлення, незважаючи на їх тривожний характер.

Будь-яке програмне забезпечення вимагає введення даних, і безпечна програма повинна розглядати всі зовнішні дані як ненадійні, якщо це не продемонстровано інакше. Коли зловмисник може надіслати виконуваний код до програми та обманом запустити його на стороні серверу або бази даних, виникають вразливості введення коду. Це дає зловмиснику можливість обійти будь-які обмеження безпеки, встановлені автором програми.

Нижче наведено кілька прикладів ін’єкції коду:

1. Ін'єкція SQL. Зловмисник може виконувати довільні оператори SQL у сприйнятливому додатку, використовуючи небезпечні параметри HTTP під час створення запитів на вебсайті.

2. Небезпечна обробка параметрів HTTP може дозволити впроваджувати шкідливий JavaScript у вебпрограму.

3. Зловмисник може використовувати небезпечну обробку параметрів HTTP для виконання довільних команд на вебсервері.

4. Ін’єкції MongoDB. Зловмисник може сконструювати запит що містить команди для керування вмістом в базі даних [17].

В ході побудови додатків для даної дипломної роботи – використовується NoSQL база даних MongoDB.

Колекції - це найближче, що має Mongo до таблиць. Це просто підсекції в більшому сегменті бази даних. Синтаксис для видалення колекції «Товарів» буде виглядати як:

db.goods.drop();

Кінцева мета зловмисника полягає в тому, щоб програма ведення записів каталогу вказувала своїй базі даних виконувати цю інструкцію.

Вебдодаток, швидше за все, дасть команду своїй базі даних запустити:

db.goods.insert({ name: «Комп’ютер»});

Щоб додати товар на ім’я «Комп’ютер». Для того, аби видалити колекцію з бази даних, зловмисник міг би передати наступні дані:

Комп’ютер»}).then(() => db.goods.drop())

Цей процес має декілька етапів:

1. Спочатку передаються дані, які очікує програма (рядок «Комп’ютер»).

2. Після чого – через ланцюжок «then» для промісів – передається наступна команда.

3. В коді колбеку - передано команду на видалення колекції товарів. Зловмисник припускає, що існує колекція під назвою «goods», до якої можливо отримати доступ. Якщо будь-яке з цих припущень невірне, ін’єкція не вдасться.

Можливо уникнути вразливостей ін’єкції коду та підвищити безпеку вебдодатків незалежно від мови, дотримуючись певних фундаментальних практик безпеки:

1. Потрібно перевіряти та пропускати через валідацію дані: перевіряти мову програми та операційну систему на наявність спеціальних символів, таких як позначки коментарів, символи завершення рядка та роздільники команд. Приймати лише ці значення, якщо програма очікує лише обмежену кількість значень, наприклад, додавши їх до білого списку або умовно ввімкнувши їх.

2. Якомога менше використовуйте функції типу eval() та еквівалентні підпрограми для необроблених даних користувача. Щоб надійно обробляти аргументи, надані користувачем та використовувати особливості мови.

3. Потрібно ставитись до всіх даних як до ненадійних: будь-які місця, де користувачі програми можуть надати або змінити дані – можуть стати потенційним вікном для надсилання зловмисного коду. Окрім очевидних векторів ін’єкції, таких як рядки запиту та форми HTML, код також можна впроваджувати за допомогою ретельно розроблених файлів даних, оновлених вручну файлів cookie та інших засобів.

4. Потрібно використовувати інструменти статичної перевірки коду, особливо для мов з динамічною типізацією, щоб знайти вразливі місця перевірки введених даних та небезпечної оцінки.

2.3 Порушення криптографічного захисту

Будь-яка інформація, яку організація не хоче, щоб широка громадськість побачила, вважається конфіденційною інформацією. Це включає інформацію про безпеку платформи (криптографічні ключі, дані облікових записів користувачів, паролі), особисту інформацію про користувачів сервісу та все, що стосується грошових операцій [18].

Існують різні форми конфіденційних даних. Наприклад, паролі слід перетворювати таким чином, щоб їх ніколи не можна було відновити у вигляді простого тексту (використовувати алгоритми хешування). Інші, такі як імена та адреси електронної пошти, швидше за все, можна зберігати як звичайний текст, але їх слід берегти як сукупність конфіденційних даних користувача.

Якщо захист конфіденційних даних налаштовано неправильно, а дані, такі як: паролі, номери кредитних карток та особиста інформація – не захищені надійними криптосистемами, зловмисники часто націлюються на них. Основною причиною витоку конфіденційних даних є криптографічний збій.

Якщо наступні умови виконуються, можливо зіткнутися з помилкою порушення криптографічного захисту [19]:

1. Конфіденційна інформація надсилається через HTTP, FTP, SMTP та інші незахищені протоколи або зберігається у відкритому тексті у базі даних.

2. Використання застарілих або небезпечних алгоритмів криптографії.

3. Повторне використання зламаних ключів або використання слабких або за замовчуванням ключів шифрування.

4. Під час обміну інформацією з сервером шифрування не застосовується, а сертифікати сервера не перевіряються.

Слабкий криптографічний захист - це шифрування, яке легко зламати. Криптографія, сама по собі, не дає належного рівня гарантій безпеки. Концептуально зламати криптографічний захист можливо, але зробити це було б надзвичайно важко і непрактично. Порушення криптографічного захисту визначається як шифрування, яке можна зламати за розумний період часу за допомогою обладнання зловмисника.

Схема загрози порушення криптографічного захисту зображена на рисунку 2.3:

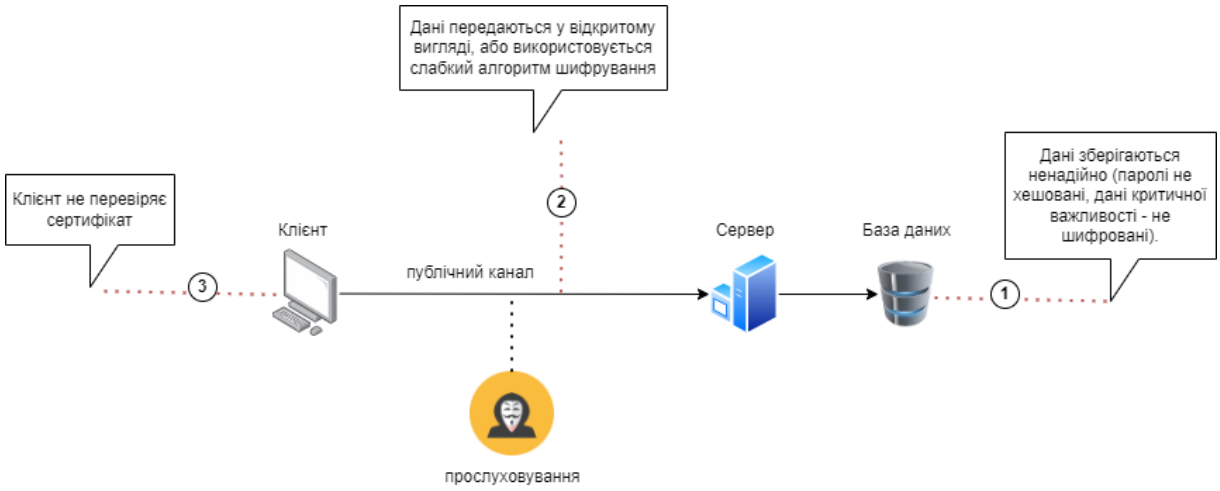

Рисунок 2.3 Схема загрози порушення криптографічного захисту

Криптографія - це широка галузь знань, і розробка безпечних криптографічних функцій може бути складною. Перше правило криптографії - ніколи не розробляти власну. За винятком експертів з інформатики, які спеціалізуються на дослідженні та створенні криптографічних функцій, все, що програміст придумає самостійно, буде принципово невірним і його буде легко зламати. Як наслідок, краще вважати, що будь-яка криптографія, розроблена власноруч, є небезпечною і її слід оновлювати.

2.4 Неправильна конфігурація безпеки

Заходи безпеки, які налаштовані неправильно - піддають ризику інформаційні системи та дані. Іншими словами, будь-які погано задокументовані зміни конфігурації, налаштування за замовчуванням або технічні збої в будь-якому компоненті кінцевих точок можуть призвести до неправильної конфігурації [20].

Деякі з найбільш поширених неправильних конфігурацій безпеки включають невимикання процесів налагодження в програмах перед розгортанням їх у виробничому середовищі, заборону програмам автоматично оновлюватися за допомогою останніх виправлень, забування вимкнути функції за замовчуванням та безліч інших незначних деталей, які можуть призвести до серйозних проблеми пізніше [21].

Якщо вразливості є точкою входу в мережу, зловмисники використовують неправильні конфігурації, щоб пробратися до бажаних цілей. Неправильну конфігурацію безпеки виправити неважко, але у великомасштабній організації вони неминучі. Їх можна знайти в будь-якому компоненті систем організації, включаючи сервери, операційні системи, програми та браузери, що робить їх пошук нетривіальною задачею. Організації вразливі до атак неправильної конфігурації через відсутність видимості та централізованих способів виправлення неправильних конфігурацій.

Визначення безпечних налаштувань для всіх програм, що розповсюджуються в компанії, є найкращим методом уникнення неправильної конфігурації безпеки. Прикладами цього є вимкнення зайвих портів, видалення програм і функцій за замовчуванням, які не потрібні основному додатку, а також вимкнення або зміна всіх користувачів і паролів за замовчуванням. Потрібно також приділяти увагу перевірці та усуненню типових помилок, наприклад, завжди вимикати режим налагодження програмного забезпечення перед випуском його у робочу станцію.

2.5 Міжсайтова підробка запиту (CSRF)

CSRF або Cross-Site Request Forgery - це атака, яка змушує автентифікованих користувачів надсилати несанкціонований запит до вебдодатку, у якому вони вже пройшли автентифікацію. Атаки CSRF використовують переваги довіри вебдодатків до користувача, який увійшов у систему. Атака CSRF використовує вразливість у вебдодатку, яка не може відрізнити запит, зроблений окремий користувач і створений без його дозволу [22, 23].

Виконання атаки міжсайтового підроблення запитів розділено на два етапи. Перший передбачає переконання жертви натиснути посилання або завантажити сторінку. Для цього зазвичай використовуються соціальна інженерія та шахрайські посилання (фішинг). Другий крок включає подання підготовленого, легального вигляду запиту на вебсайт із браузера жертви. Зловмисник вибирає значення для запиту, які можуть включати будь-які файли cookie, які жертва пов’язала з цим вебсайтом. В результаті вебсайт знає, що ця жертва має право виконувати певні функції на сервері. Навіть якщо жертва надсилає запит за ініціативою зловмисника, будьякий запит, надісланий з цими обліковими даними HTTP або файлами cookie, вважатиметься справжнім [23].

Схема роботи CSRF зображена на рисунку 2.4:

Рисунок 2.4 Схема роботи CSRF

Коли робиться запит до вебсайту, браузер жертви перевіряє, чи є в ньому файли cookie, пов’язані з джерелом вебсайту, які потрібно передати разом із запитом HTTP. Якщо це так, ці файли cookie будуть включені в кожен запит, надісланий на цей вебсайт. Дані автентифікації часто зберігаються у значенні cookie, і такі файли cookie відображають сеанс користувача. Це зроблено, щоб надати користувачеві послідовний користувацький досвід і позбавити його від необхідності підтверджувати особу щоразу, коли він відвідує нову сторінку. Якщо вебсайт приймає файли cookie сеансу і вважає, що сеанс користувача все ще дійсний, зловмисник може використовувати CSRF для надсилання запитів так, ніби вони надходять від жертви. Оскільки запити завжди надходять із браузера жертви за допомогою власних файлів cookie, вебсайт не може відрізнити запити, надіслані зловмисником, і запити, надіслані жертвою. Атака CSRF просто використовує той факт, що браузер автоматично надсилає файл cookie на сервер із кожним запитом.

Якщо жертва автентифікована, підробка міжсайтових запитів буде ефективною. Щоб атака була успішною, жертва має увійти в систему. Оскільки атаки CSRF використовуються для обходу процедури аутентифікації, деякі аспекти, такі як загальнодоступний вміст, можуть не вплинути на стан безпеки, навіть якщо вони не захищені від них. Загальнодоступна контактна форма на вебсайті, наприклад, захищена від загрози CSRF через вищевказану причину. Для подання форми такі HTML-форми не вимагають від жертви будь-яких привілеїв. CSRF застосовується лише тоді, коли жертва має можливість виконувати дії, доступні не всім.

Під час успішної атаки CSRF зловмисник змушує користувача-жертву випадково виконати дію. Це може бути, наприклад, оновлення адреси електронної пошти облікового запису, скидання пароля або здійснення грошового переказу. Зловмисник може отримати повний контроль над обліковим записом користувача залежно від характеру дії. Якщо скомпрометований користувач має привілейовану роль у програмі, зловмисник може отримати повний контроль над даними та функціями.

Висновки за розділом 2

На сьогодні, люди настільки звикли отримувати майже будь-яку послугу чи інформацію онлайн, що важко уявити сучасну компанію без вебдодатка. Завдяки цьому, вимоги до якості вебдодатка, стабільності його роботи, а головне, належного рівня безпеки - надзвичайно високі.

Ризики безпеки у вебдодатках стають однією з найактуальніших проблем. Якщо зловмисники отримають доступ до конфіденційних даних, компанії можуть зазнати серйозних фінансових і репутаційних втрат, тим більше, що список потенційних недоліків безпеки сайту досить довгий.

Вразливості вебдодатків включають системний недолік або слабкість у вебдодатку. Вони існують протягом багатьох років, здебільшого через те, що вони не перевіряють або не очищають введені форми, неправильно налаштовані вебсервери та недоліки дизайну додатків і їх можна використовувати, щоб поставити під загрозу безпеку програми.

Існують рішення безпеки вебдодатків, розроблені спеціально для програм і тому важливо виходити за рамки традиційних сканерів вразливостей, коли справа доходить до виявлення прогалин у безпеці програм організації. Щоб по-справжньому зрозуміти свої ризики, потрібно дізнатись більше про деякі типи вебдодатків і атак на кібербезпеку.

РОЗДІЛ 3
ПРОГРАМНА ІМПЛЕМЕНТАЦІЯ ВЕБДОДАТКУ З ВИКОРИСТАННЯМ КОМБІНАЦІЙ МЕХАНІЗМІВ ЗАХИСТУ

3.1 Вибір стеку технологій та його огляд

Будь-яка компанія-розробник повинна інвестувати в розробку сучасних програмних додатків. Постійні вдосконалення технології розробки додатків надали власникам бізнесу та керівникам проектів безліч варіантів на вибір. З іншого боку, різноманітність доступних технологій може лякати. Дуже важливо вибрати правильний IT-стек для компанії залежно від цілей і пріоритетів.

Node.js є однією з таких технологій, яка широко використовується в наш час. Багато компаній використовували Node.js для створення свого внутрішнього технологічного стека, оскільки це сучасна платформа додатків. Його популярність як платформи для розробки додатків на стороні сервера можна побачити в усіх секторах.

Node.js - це цікаве поєднання технологій фронтенду та бекенда з точки зору стеків додатків. Node.js розширює можливості JavaScript, який, як правило, є інтерфейсною мовою вебскриптів на стороні клієнта, для роботи на серверній частині архітектури вебдодатків, а також для безсерверної архітектури [24].

Node.js досягає цього, запускаючи на сервері в межах власного середовища виконання. Середовище виконання Node.js спроектовано легким і ефективним, з неблокуючим введенням-виводом і керуванням пакетами, щоб зробити розробку додатків Node.js ще простішою.

Нижче наведено найважливіші переваги Node [25]:

1. Пришвидшує розробку додатків у режимі реального часу з високим трафіком за рахунок зрозумілого синтаксису та відсутності великої кількості повторюваних шаблонів коду.

2. Дозволяє писати код JavaScript як на стороні клієнта, так і на стороні сервера.

3. Робить процес розробки більш ефективним, усуваючи розрив між інженерами інтерфейсу та бекенда.

4. Постійно розширювана база пакетів надає розробникам доступ до різноманітних інструментів і модулів, що дозволяє їм працювати ефективніше.

5. Код працює швидше, ніж код, написаний будь-якою іншою мовою за рахунок асинхронності виконання завдань вводу-виводу.

6. Node ідеально підходить для мікросервісів, які є популярним рішенням для корпоративних додатків.

Недоліки використання серидовища:

1. Важкі обчислювальні завдання можуть викликати вузькі місця в продуктивності.

2. Проблема зі зворотними викликами функцій.

3. Незрілість інструментарію.

4. Зростає попит на кваліфікованих працівників.

Node.js - це надійна технологія, яка добре працює в багатьох ситуаціях. Кількість прикладів додатків Node.js велика, і ця технологія може бути корисною для великої кількості комерційних проектів.

Крім того, велика спільнота Node.js постійно намагається покращити платформу і цілком можливо, що навіть сценарії, коли Node.js не є найкращим рішенням, застаріють.

Express.js - це серверна платформа для онлайн-додатків Node.js, яка оптимізована для створення односторінкових, багатосторінкових і гібридних вебпрограм. Це стало стандартною серверною системою для node.js. Стек MEAN складається з кількох компонентів, одним з яких є експрес.

Express.js дозволяє легко створювати різні типи вебпрограм за короткий проміжок часу. Для запитів клієнтів Express.js забезпечує просту систему

маршрутизації. Він також включає проміжне програмне забезпечення, яке відповідає за прийняття рішень, щоб запропонувати правильні відповіді на запити клієнта.

Щоб створити компонент маршрутизації без Express.js - потрібно написати власний код, що займає багато часу, займає багато місця і важко читається. Express.js надає програмістам простоту, гнучкість, ефективність, мінімалізм і масштабованість. Оскільки це фреймворк Node.js, він також пропонує переваги високої продуктивності.

За допомогою циклу подій у Node.js всі задачі виконуються надзвичайно швидко, уникаючи будь-якої неефективності. Найпопулярнішими аспектами, які цінуються розробниками вебдодатків, є чудова продуктивність Node.js і простота написання сценаріїв у Express.js.

Програмний стек MEAN - це безкоштовний стек програмного забезпечення JavaScript з відкритим вихідним кодом для створення інтерактивних вебсторінок і вебпрограм. ExpressJS - це попередньо вбудований фреймворк Node.js, який може допомогти розробляти швидші та більш прості для розуміння вебдодатки на стороні сервера.

MongoDB - це документно-орієнтована база даних з динамічною структурою, яка зберігає дані в JSON-подібних документах. Це означає, що і ній немає чітких і строгих вимог щодо створення і дотримання єдиної структури даних, наприклад, кількість полів або типи полів, які використовуються для зберігання значень. Об’єкти JSON подібні до документів MongoDB [26].

Переваги MongoDB:

1. Це база даних NoSQL без схеми. Працюючи з MongoDB - не доведеться турбуватися про розробку схеми бази даних.

2. Ефективно підтримує геопросторові дані та функції.

3. Гнучкість вмісту полів документа.

4. Дозволяє ACID переходи між кількома документами.

5. Має високий рівень продуктивності, доступності та масштабованості.

3.2 Розробка системи аутентифікації

Між пристроєм користувача та сервером вебпрограм надсилається велика кількість даних, зокрема інформації, яка потребує аутентифікованого доступу, що робить безпеку ще більш важливою, ніж будь-коли.

Аутентифікація є загальним підходом для всіх програм для забезпечення безпеки. Це єдиний спосіб відповісти на запит програми щодо рідтвердження особи користувача. Коли справа доходить до архітектури без збереження стану або архітектури, орієнтованої на обслуговування, на ринку з’являється багато нових концепцій і рішень.

Існують три найбільш поширені методи здійснення аутентифікації:

1. Аутентифікація за допомогою файлів cookie.
2. Аутентифікація за допомогою токенів.
3. Залучення третьої сторони типу OAuth.

Для виконання проектів дипломної роботи, було використано два методи аутентифікації користувача: за допомогою файлів cookie та JWT токенів.

Основні рекомендації, яких слід дотримуватися при виборі методу:

1. Аутентифікація на основі сеансу за іменем користувача та паролем часто є найбільш підходящою для вебпрограм, які використовують шаблони на стороні сервера.

2. Аутентифікація на основі токенів є кращим методом для RESTful API, оскільки вони не мають стану.

3. Якщо є потреба у підвищеній безпеці та надійності – варто скористатись послугами третьої сторони, які також можуть надавати послуги багатофакторної аутентифікації.

3.2.1 Розробка системи аутентифікації на основі сесії

Протокол HTTP є основою вебдодатків. HTTP - це протокол без стану, що означає, що клієнт і сервер забувають один про одного в кінці кожного циклу запиту та відповіді.

Для вирішення цієї проблеми – було придумано поняття сесії. Щоб дозволити серверу відстежувати стан користувача, кожен сеанс буде нести деякі унікальні дані про цього клієнта. Стан користувача зберігається в пам'яті сервера або в базі даних під час аутентифікації на основі сеансу.

Схема роботи аутентифікації на основі сесії зображена на рисунку 3.1:

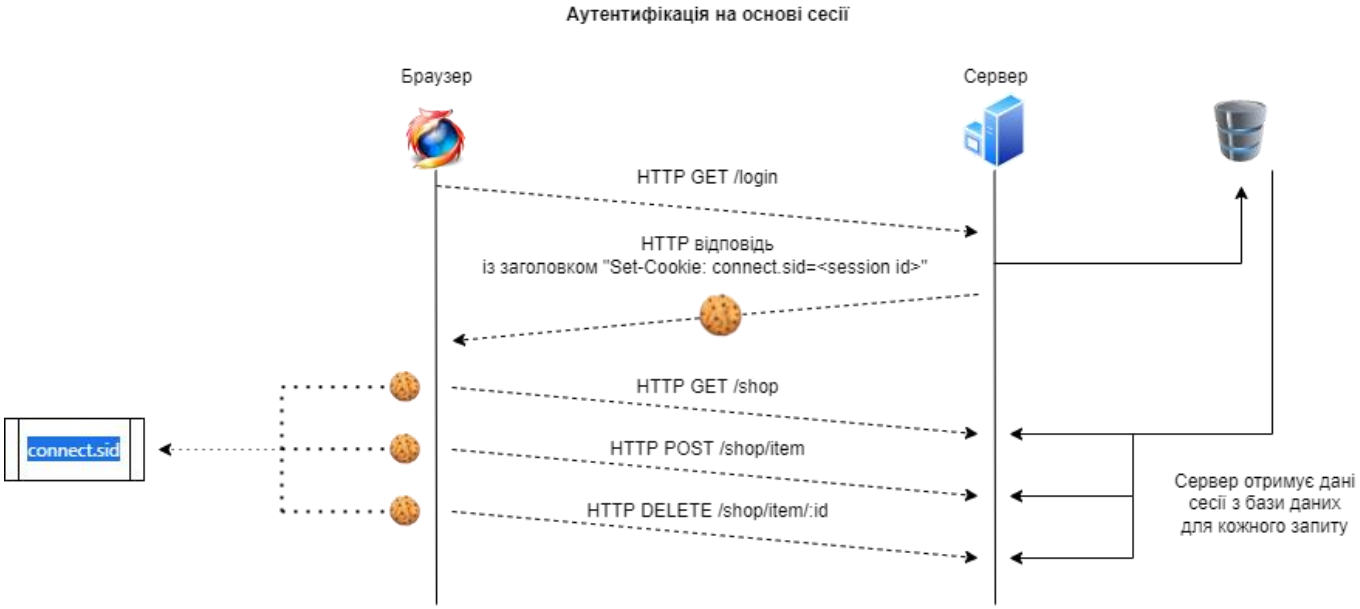

Рисунок 3.1 Схема роботи аутентифікації на основі сесії

Коли клієнт виконує запит на вхід, сервер створює сеанс і зберігає його в базі даних. Файл cookie надсилається сервером, коли він відповідає клієнту. Цей файл cookie буде містити унікальний ідентифікатор сеансу, який раніше зберігався на сервері, а тепер зберігається на клієнті. Кожен раз, коли клієнт робить запит до сервера, цей файл cookie буде передаватися [27].

Щоб зберегти відповідність один до одного між сеансом і файлом cookie, сервер використовує цей ідентифікатор для пошуку сеансу, який зберігається в базі

даних або сховищі сеансів. В результаті з'єднання протоколу HTTP будуть мати стан.

Переваги використання сесій для аутентифікації:

1. Оскільки облікові дані не потрібні, наступний вхід відбувається швидше.

2. Користувальницький досвід покращено.

3. Це просто втілити в дію. Багато фреймворків мають цю функцію за замовчуванням.

Недоліки використання сесій для аутентифікації:

1. Стан зберігається в єдиному місці, тому для розподілених додатків – це може створювати складнощі.

2. Навіть якщо запит не вимагає аутентифікації, файли cookie передаються разом із ним.

3. Можливі атаки CSRF.

Файл cookie – це пара ключ-значення, яку зберігає браузер. Кожен запит HTTP, що надходить на сервер, супроводжується файлами cookie з браузера.

Сервер не можете зберегти багато інформації в файлі cookie. Будь-які облікові дані користувача або секретна інформація не можуть зберігатися в файлі cookie. Якщо це зробити, хакер міг би просто отримати доступ до цієї інформації та використати її у злих цілях.

Дані сеансу, з іншого боку, зберігаються на стороні сервера, в базі даних або сховищі сеансів. В результаті він може обробляти більші обсяги даних. Сеанс аутентифікується за допомогою секретного ключа або ідентифікатора сеансу, який сервер отримує від файлу cookie під час кожного запиту на доступ до даних.

3.2.2 Розробка системи аутентифікації на основі токенів доступу

JSON Web Token (JWT) - це стандарт, який визначає стислий і автономний метод для безпечного обміну інформацією як об'єкта JSON між сторонами. Через їх малий розмір, токени легко надсилати через URL-адресу, параметр POST або

заголовок HTTP. Для цифрового підпису інформації в JWT використовується пара секретних або відкритих/приватних ключів. Для підписання JWT можна використовувати секретний ключ [28].

Аутентифікація є основним використанням JWT. JWT призначається користувачеві після входу в програму. Подальші запити користувача включатимуть даний JWT. Цей токен інформує сервер про маршрути, служби та ресурси, які користувач має дозвіл використовувати.

JWT токен зображено на рисунку 3.2:

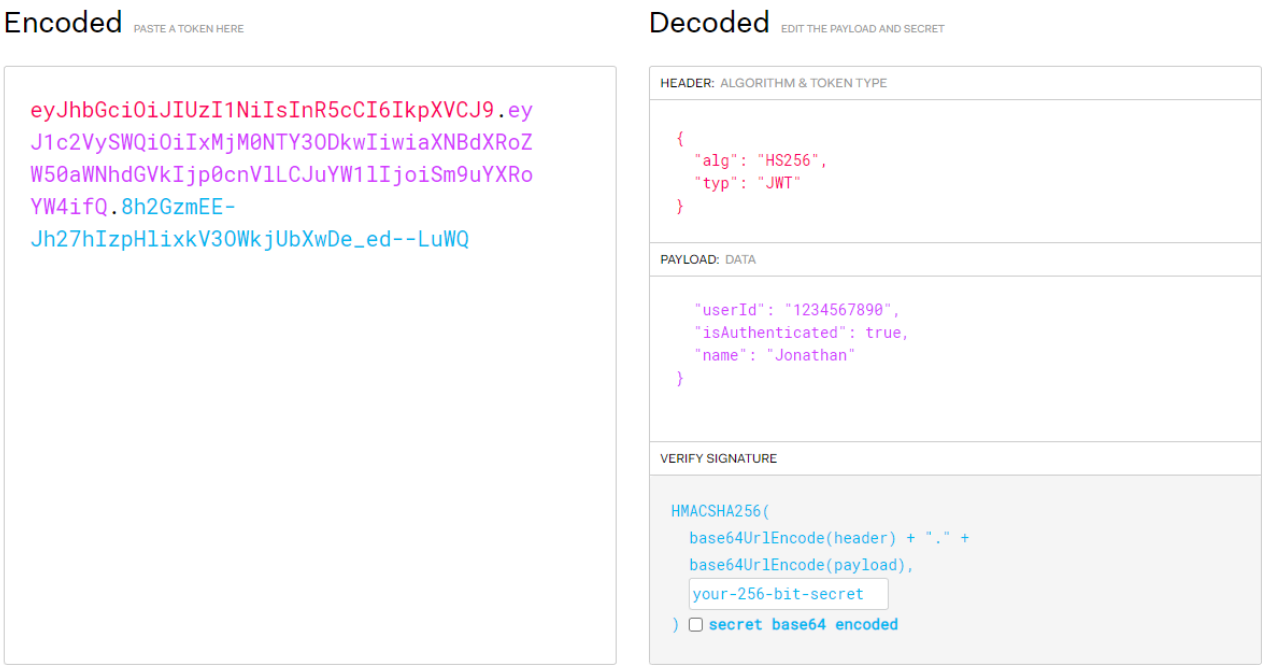

Рисунок 3.2 JWT токен

JWT є особливо корисним інструментом для автентифікації API та дозволу між серверами. Єдиний вхід - це можливість постачальника послуг отримувати надійну інформацію від сервера аутентифікації.

Постачальник послуг може хешувати частину отриманого токена та порівняти його з підписом токена, надавши секретний ключ постачальнику ідентифікаційних даних. Таким чином, якщо розподілити ключ між різними частинами мікросервісної архітектури, кожний сервіс додатку зможе надійно перевіряти стан аутентифікації користувача [28].

Схема роботи аутентифікації на основі JWT токенів зображена на рисунку 3.3:

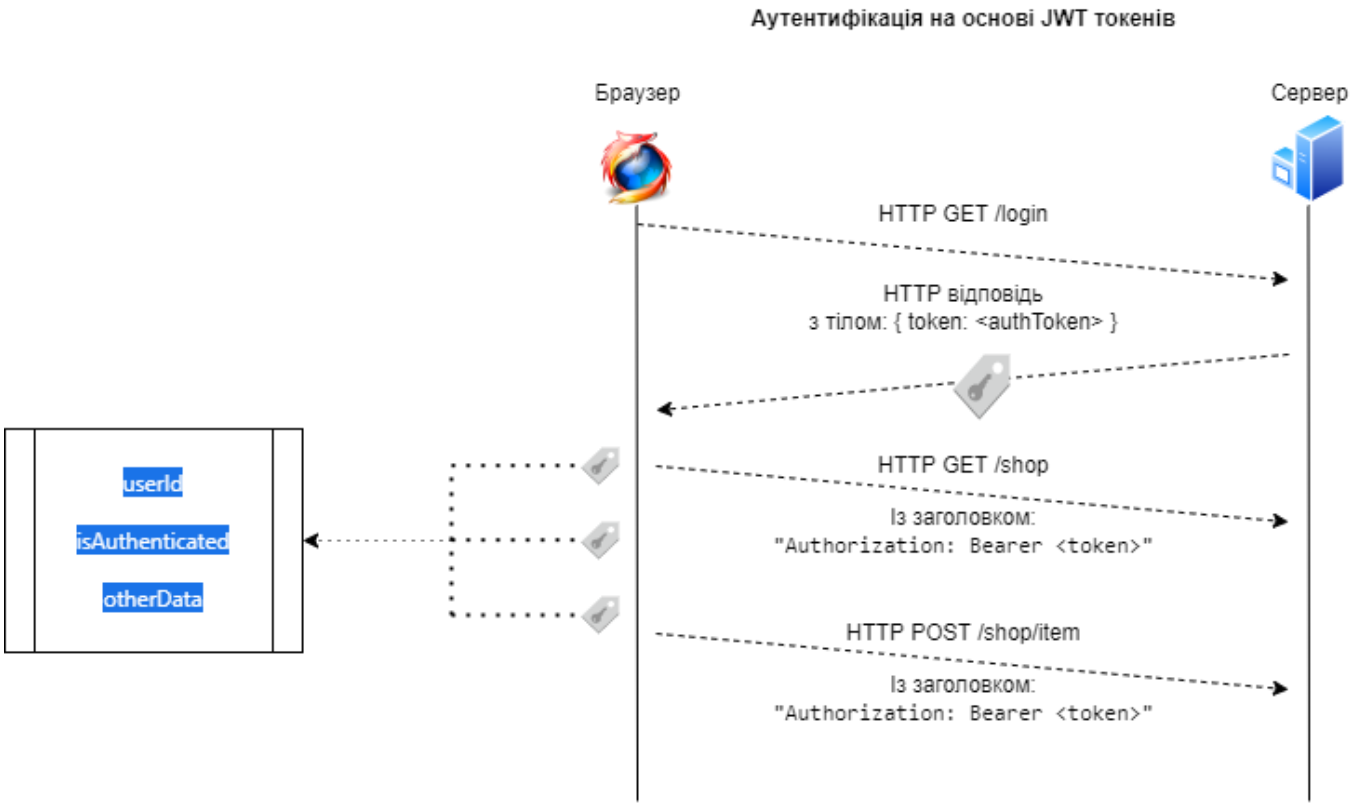

Рисунок 3.3 Схема роботи аутентифікації на JWT токенів

Щоб перевірити JWT, сервер повторно генерує підпис, використовуючи заголовок і корисне навантаження одержуваного JWT, а також секретний ключ. JWT вважається дійсним, якщо щойно виготовлений підпис збігається з підписом на JWT.

Якщо зловмисник буде намагатися створити власний токен, або модифікувати існуючий, він можете легко створити заголовок і корисне навантаження, але він не зможете згенерувати дійсний підпис, не знаючи ключа. Якщо він змінить наявне корисне навантаження JWT, підписи більше не збігатимуться.

Переваги використання JWT для аутентифікації:

1. У ньому немає стану. Оскільки токен можна перевірити за допомогою підпису, серверу не потрібно його зберігати. Це прискорює запит, усуваючи необхідність пошуку в базі даних.

2. Розроблено для архітектур мікросервісів із багатьма службами, які потребують аутентифікації. На кожному кінці потрібно лише налаштувати, як працювати з токеном і його ключем.

Недоліки використання JWT для аутентифікації:

1. Атаки XSS через localStorage або CSRF через файли cookie можуть відбуватися залежно від того, як токен зберігається на клієнті.

2. Токени не підлягають відкликанню. Їх можна використовувати лише один раз. Це означає, що якщо токен скомпроментовано, зловмисник зможе використовувати його до закінчення терміну його дії.

3. Токени оновлення мають бути налаштовані на автоматичну видачу токенів після закінчення терміну їх дії.

Однак JWT не слід використовувати як токени сеансу за замовчуванням. JWT має величезну кількість функцій і широке охоплення, що підвищує ризик помилок авторів або користувачів бібліотеки.

Інша складність полягає в тому, що, оскільки JWT є автономними і не мають центрального повноваження для їх визнання недійсними, сервер не може видалити їх після завершення сеансу. На додачу, JWT досить великі і займають трафік.

JWT є широко використовуваним стандартом, що працює завдяки покладанню на підписи для довіри запитів та обміну даними між сторонами. Перед застосуванням, потрібно переконатись, що команда має розуміння, коли його найкраще використовувати, коли найкраще уникати та як уникнути найбільш фундаментальних ризиків безпеки.

3.3 Розробка системи безпечної заміни паролю

Придумати надійний пароль, який можна запам’ятати, непросто. Користувачі, які використовують кілька паролів для різних вебсайтів, більш схильні їх забути. Тому дуже важливо мати інструмент, який дозволить вашим користувачам безпечно скинути свій пароль, якщо вони його забудуть.

Відновлення паролю - це одна з основних функцій, яку мають всі веб/мобільні програми. Одна з обов’язкових речей для реалізації цього - у користувача має бути налаштована адреса електронної пошти під час реєстрації або у профілі користувача, щоб можна було надіслати посилання на електронну пошту для відновлення пароля.

Процедура відновлення паролю складається з наступних етапів:

1. Користувач переходить на сторінку скидання пароля, де він повинен ввести свою адресу електронної пошти. Після введення адреси електронної пошти серверу надсилається запит, щоб перевірити, чи ця адреса електронної пошти вже існує в будь-якому профілі користувача.

Сторінка зі скиданням паролю зображена на рисунку 3.4:

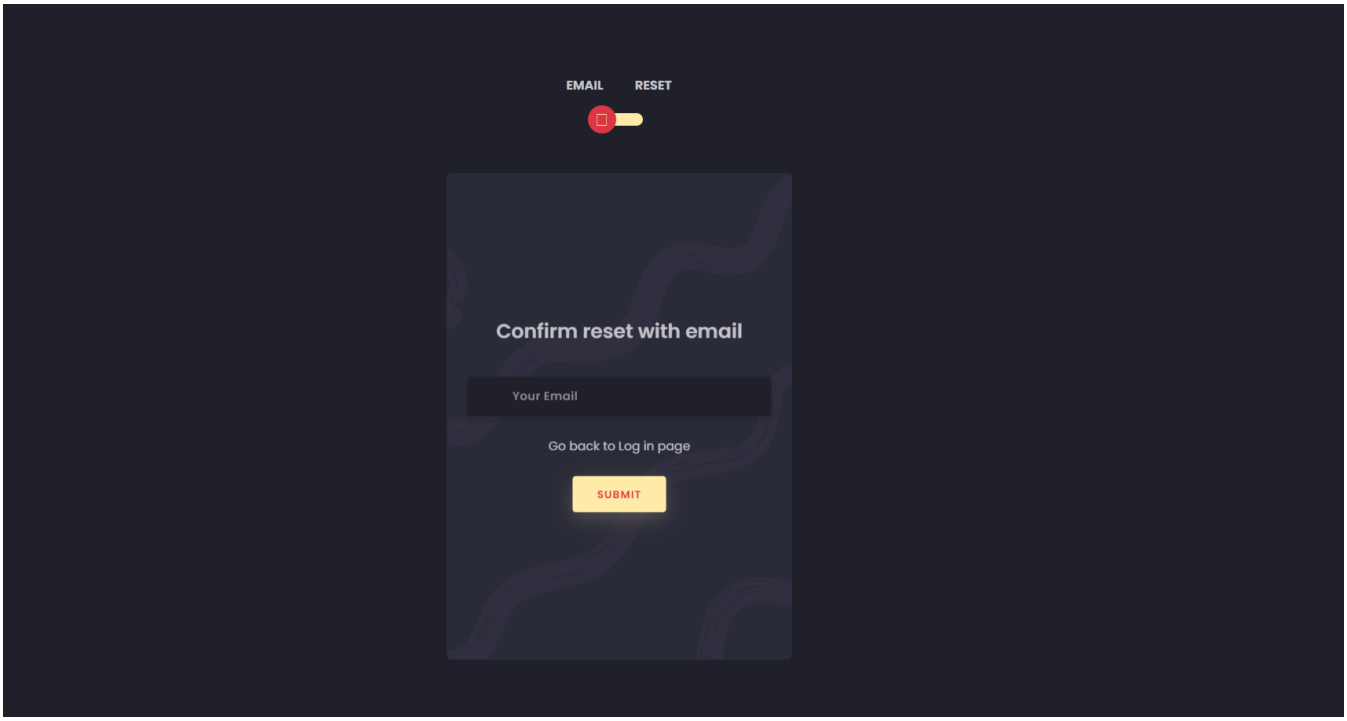

Рисунок 3.4 Сторінка зі скиданням паролю

2. Електронна пошта, надіслана зі сторінки відновлення пароля, перевіряється, чи є вона в базі даних. Якщо адреса електронної пошти не збігається, буде надіслано відповідь із помилкою.

3. Якщо пошта наявна в базі даних – сервер генерує токен, за допомогою функції, що надає випадкове число фіксованої кількості біт (бажано не менше 128) та зберігає цей токен в моделі користувача в базі даних.

4. Сервер створює електронний лист адресований на введену пошту та вставляє в неї згенероване посилання вигляду: «https://server.com:443/api/auth/reset/${userId}/${token}». Де «${token}» - попередньо збережене в моделі користувача випадкове число.

На рисунку 3.5 – показане посилання, згенероване для користувача:

Рисунок 3.5 Посилання, згенероване для користувача

5. Коли користувач отримує лист – він переходить за наданим посиланням, тож його браузер надсилає GET запит за вказаною адресою.

На рисунку 3.6 зображено вигляд такого листа, що надсилається користувачеві:

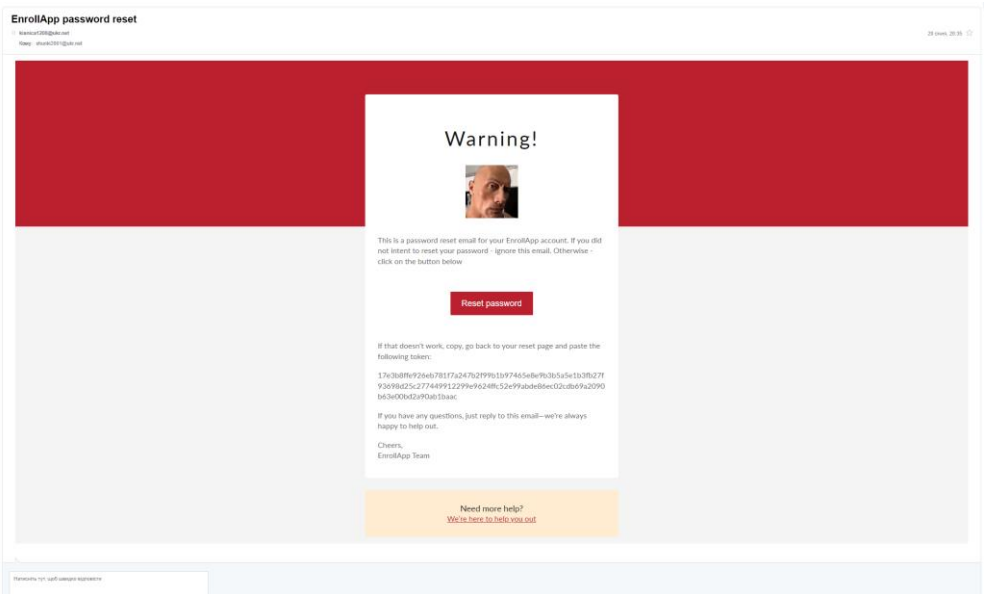

Рисунок 3.6 Лист, надісланий користувачеві

Коли користувач натискає посилання - він потрапляє на сторінку, де можна скинути пароль. Оскільки це просто API - можливо використовувати будь-яку технологію для створення інтерфейсу.

На цьому етапі вже надається токен та ідентифікатор користувача, щоб підтвердити користувача та створити новий пароль.

6. Сервер отримує запит та дістає з URL значення токену. Далі, знаходить користувача в базі даних за userId та перевіряє токен і якщо вони срівпадають – надає сторінку, де можливо ввести новий пароль для користувача.

На рисунку 3.7 зображено сторінку заміни паролю:

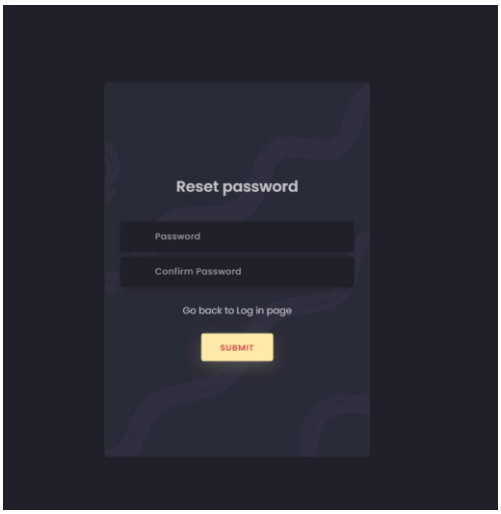

Рисунок 3.7 Сторінка заміни паролю

Надійна процедура скидання пароля досягає двох цілей:

1. Вони зводять труднощі клієнта до мінімуму. Споживач повинен мати можливість скинути свій пароль менш ніж за хвилину і процедура має включати лише інформацію, яку йому зручно вводити, наприклад адреси електронної пошти.

2. Вони гарантують, що інформація клієнтів зберігається в безпеці. Забезпечення захисту від таких речей, як багаторазові невдалий вхід та передача даних лише через захищені мережі.

Оскільки електронна пошта відповідає обом цим умовам, електронна пошта є найбільш часто використовуваним методом для скидання пароля. Це зменшує тертя, дозволяючи клієнтам швидко та легко вводити свою адресу електронної пошти, а

також захищає їх інформацію, дозволяючи лише клієнтам доступ до своєї поштової скриньки.

3.4 Застосування можливостей протоколів прикладного рівня для керування безпекою

Express, як правило, надійний для повноцінної розробки додатків і створення прототипів, однак - він не позбавлений недоліків, коли справа доходить до безпеки. Причина цього в тому, що Express призначений для створення онлайн-додатків, а не для захисту сервера Node.js від недоліків безпеки. Заголовки HTTP в основному відкриті за замовчуванням, що сприяє швидкій розробці додатків [29].

Оскільки заголовки HTTP не видно користувачам, розробники, як правило, нехтують ними. Заголовки HTTP, з іншого боку, можуть помилково відкрити важливу інформацію про програму, тому дуже важливо налаштувати та використовувати їх безпечно.

Заголовок X-Powered-By є звичайним механізмом витоку інформації програмами Express. Цей заголовок повідомляє вебпереглядачу, який фреймворк використовується на сервері та яка версія використовується. Express додає заголовок X-Powered-By і вписує як значення «Express» за замовчуванням, без номера версії. Хакери часто посилаються на цю інформацію зі списком загальновизнаних відомих уразливостей, що робить програму мішенню для простих атак, особливо, якщо використовується версія Express без виправлень знайдених вразливостей.

Захищаючи заголовки HTTP, які повертаються додатками Express, Helmet.js долає розрив між Node.js і Express.js. HTTP за своєю суттю небезпечний і відкритий. Він може розкривати важливу інформацію про програму та відкривати дані будь-кому, хто має базові комп’ютерні знання та вміння.

Використання заголовків HTTP за замовчуванням є надійним методом швидкого запуску програми, але це ризикує надати будь-кому небажаний доступ до неї. Що ще гірше, оскільки більшість кінцевих користувачів не усвідомлюють

важливості HTTPS над HTTP, розробники, як правило, нехтують ним, поки їх політика безпеки не вимагатиме від них використання HTTPS.

Helmet.js спрощує для розробників Node.js захист HTTP-заголовків. Helmet.js - це набір з 12 модулів, які працюють разом для підключення до Express. Параметри конфігурації для захисту різних заголовків HTTP доступні в кожному модулі.

Helmet.js містить набір модулів Node, які можна використовувати для підключення до Express і покращення безпеки заголовка HTTP. Який механізм стоїть за цим? Він дозволяє встановлювати заголовки та запобігати типовим уразливостям, таким як клікджекінг, жорстка реалізація HTTP та параметри завантаження для вразливих браузерів, таких як Internet Explorer 8.

Можливо використовувати Content-Security-Policy з Helmet.js, щоб змусити наступних розробників працювати над відкритими API, яким потрібен HTTP, щоб підходити до коду з безпекою в першу чергу.

Helmet.js містить додаткові вбудовані модулі для підвищення безпеки програми Express [29, 30]:

1. Content-Security-Policy - дозволяє встановити Content-Security-Policy для захисту від міжсайтових скриптових атак..
2. Expect-CT - це програма, яка керує прозорістю сертифікатів.
3. X-DNS-Prefetch-Control - використовується для керування отриманням DNS браузера.
4. X-Frame-Options - для захисту від clickjacking.
5. X-Powered-By - для видалення заголовку «X-Powered-By».
6. Закріплення відкритого ключа HTTP здійснюється за допомогою Public-Key-Pins.
7. Strict-Transport-Security: - це політика для HTTP Strict Transport.
8. X-Download-Options - обмежує параметри завантаження.
9. Cache-Control - вимкнення кешування на стороні клієнта.
10. X-Content-Type-Options - використовується для захисту від атаки Sniffing.
11. Referrer-Policy - ховає заголовок «Referrer».
12. X-XSS-Protection - використовується для захисту від атак XSS.

На рисунку 3.8 зображено заголовки відповіді сервера:

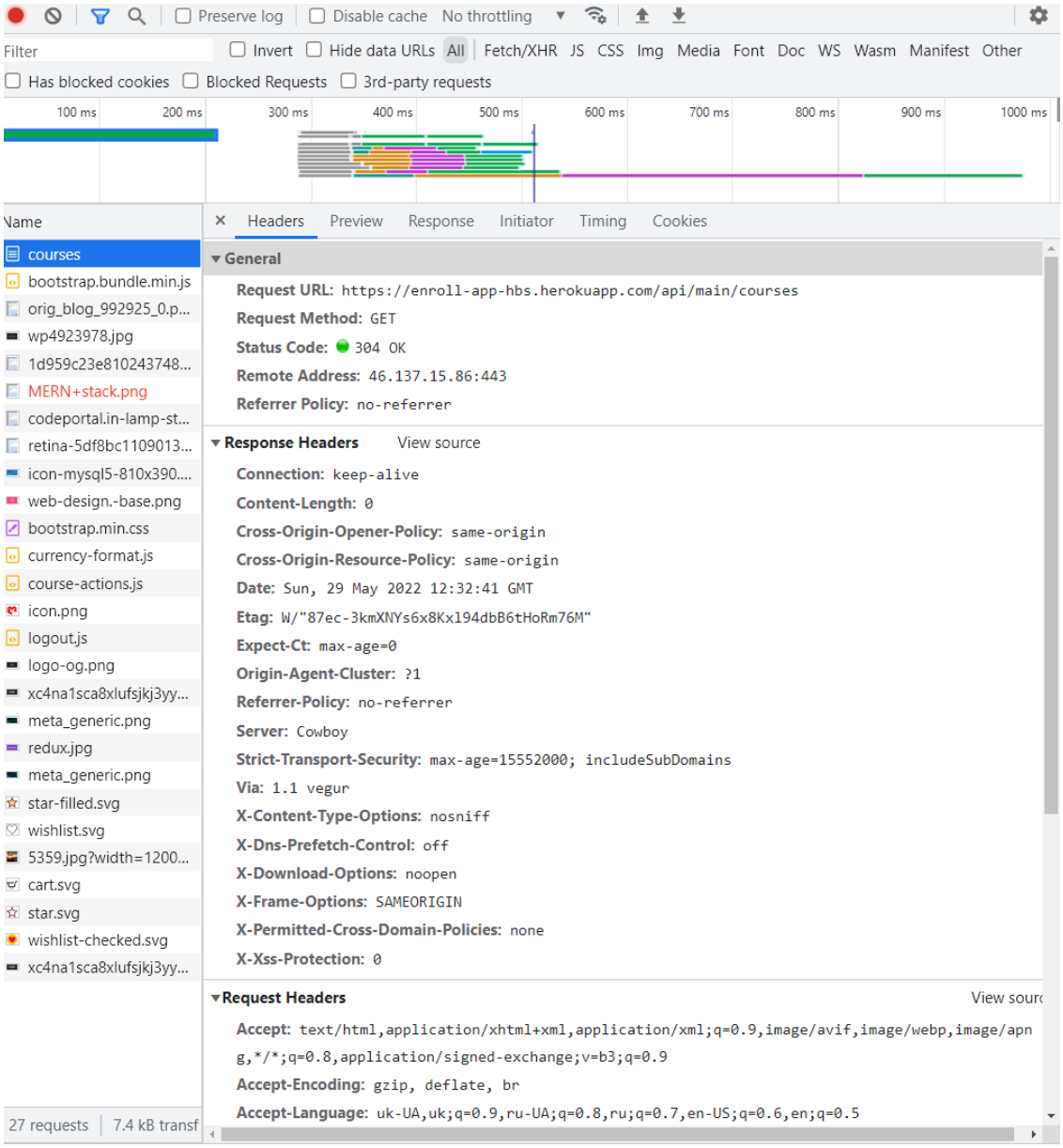

Рисунок 3.8 Заголовки відповіді сервера

Перехід від стандартних налаштувань Express до більш безпечного набору за замовчуванням за допомогою Helmet.js - це швидкий і простий підхід, щоб додати рівень безпеки до програми Express. Крім того, Helmet.js дозволяє легко налаштувати HTTP-заголовки за допомогою доступних модулів.

3.5 Застосування механізмів захисту від CSRF

Атак CSRF можна уникнути, використовуючи токени CSRF. Процедура передбачає такі етапи [31]:

1. Сервер надсилає токен клієнту.
2. Клієнт заповнює форму та надсилає її разом із токеном.
3. Якщо токен недійсний, сервер відхилить запит.

Токени CSRF є одними з найбільш широко використовуваних і схвалених методів захисту проти CSRF. Токен CSRF підтримується всіма сучасними мовами програмування вебдодатків.

На рисунку 3.10 зображено використання CSRF токену:

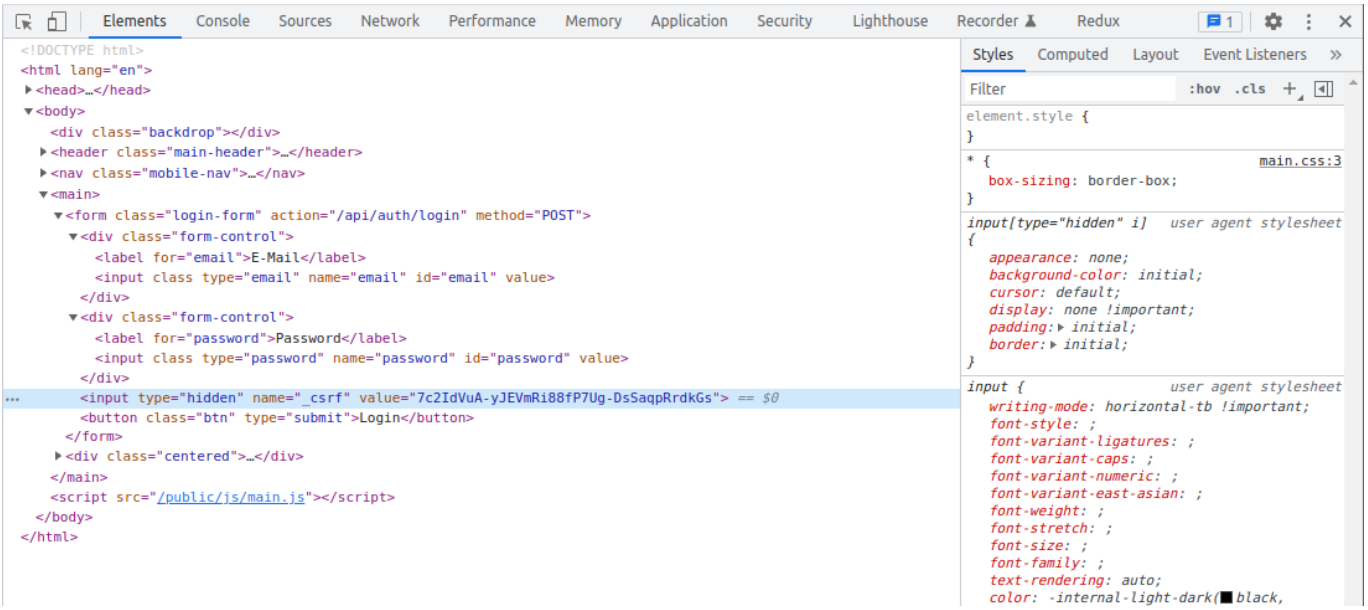

Рисунок 3.10 Використання CSRF токену

CSURF - це офіційне проміжне програмне забезпечення захисту CSRF для Node.js.

1. Для початку, потрібно розробити проміжне програмне забезпечення для виробництва та перевірки токенів CSRF.

2. Під час запиту – токен зберігається у об’єкті відповіді сервера та є доступним за допомогою модуля cookie-parser.

3. Використовуючи проміжне програмне забезпечення - потрібно передати токен як приховане значення в формі.

4. Для перевірки токену використовується автоматично налаштований middleware, який перевіря тіло запиту на наявність токену.

Зловмиснику доведеться якимось чином отримати токен CSRF з вашого сайту, і йому доведеться це зробити за допомогою JavaScript. В результаті, якщо ваш сайт не підтримує CORS, зловмисник не зможе отримати токен CSRF, таким чином усуваючи загрозу.

Потрібно переконатися, що AJAX не має доступу до токенів CSRF. Не потрібно створювати маршрут /csrf виключно для отримання токена, а також підтримувати CORS на цьому маршруті.

Необхідно лише, щоб токен був сформованим надійним генератором псевдопослідовностей, що ускладнює зловмиснику вгадати за кілька спроб. Він не повинен бути безпечним з точки зору криптографії. Атака визначається як один або два клацання невідомим користувачем, а не груба атака сервера.

CSRF стає все менше великою загрозою, оскільки Інтернет розвивається в напрямку API, а браузери стають більш безпечними завдяки додатковим правилам безпеки. Потрібно блокувати відвідування сайту старішим браузерам і конвертувати якомога більше своїх API в JSON API, де токени CSRF фактично застаріли і не мають сенсу.

3.6 Застосування системи перевірки вхідних даних

Перевірка якості, чіткості та специфіки даних є важливою для пом’якшення недоліків проекту. Команда ризикує приймати рішення на основі даних, які мають недоліки і не вказують на поточну ситуацію, якщо вони не перевіряють їх.

Хоча дуже важливо перевірити введені дані та значення, модель даних також має бути перевірена. При спробі використовувати файли даних у різних програмах і програмному забезпеченні , можливо зіткнутись з проблемами, якщо модель даних не буде структурована або розроблена неправильно.

Те, що програміст може робити з даними, визначається форматом і вмістом файлів даних. Правила перевірки використовуються для очищення даних перед їх використанням, що допомагає уникнути проблем. Забезпечення достовірності результатів починається із забезпечення цілісності даних.

Перевірка введення на стороні клієнта обробляється зовнішнім кодом, тоді як перевірка введення на стороні сервера обробляється внутрішнім кодом. Оскільки внутрішній код найбільш тісно пов’язаний з базою даних, він діє як міст між зовнішнім кодом, який бачить користувач, і базою даних, в якій зберігаються дані користувача.

Під час розробки програм, особливо для клієнтських, перевірка на стороні сервера має вирішальне значення. Причина цього полягає в тому, що ніколи не слід покладатися виключно на введення користувача, оскільки такі введення іноді можуть містити неправдиву або шкідливу інформацію [32].

На рисунку 3.10 зображено валідацію на стороні клієнта:

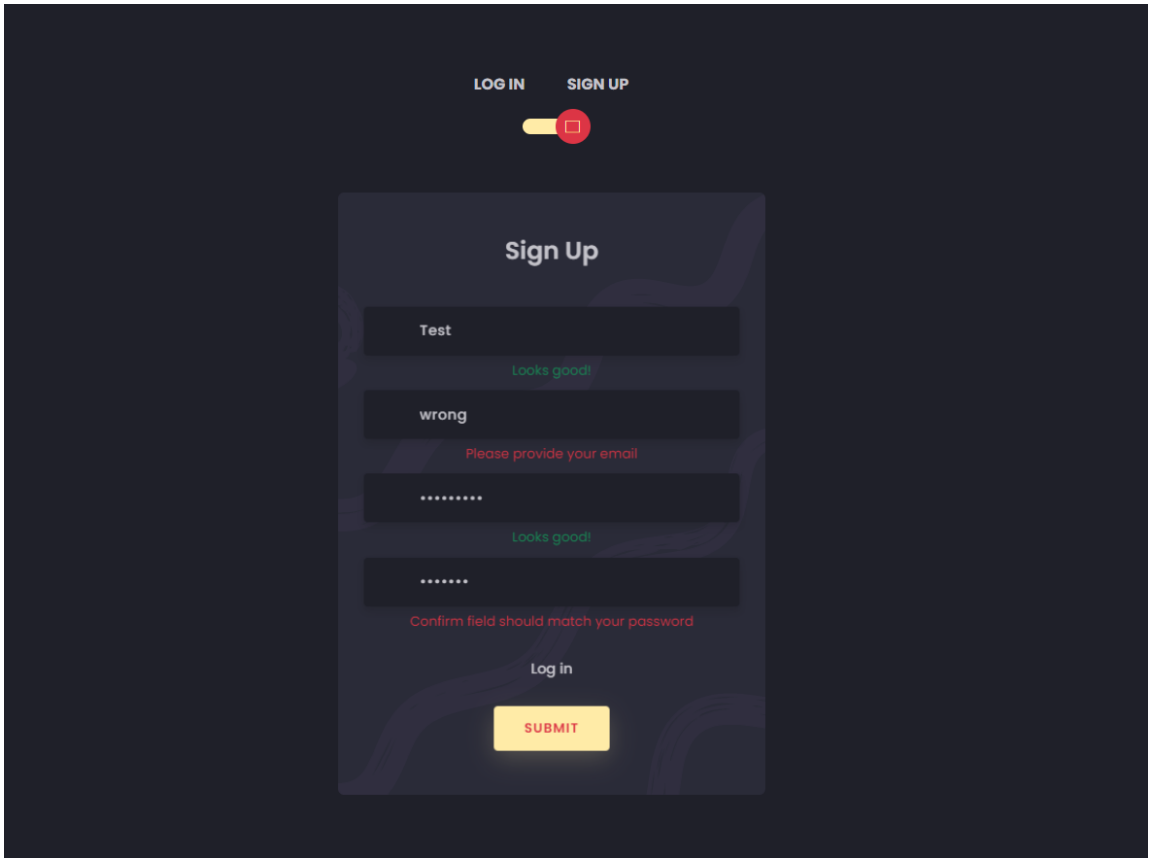

Рисунок 3.10 Валідація на стороні клієнта

Перевірка на стороні клієнта є відмінним підходом для відфільтрування більшості даних, але перевірка на стороні сервера все ще потрібна, оскільки API сервера доступно не тільки через клієнтську програму [32].

Існують різні способи перевірки даних у Node.js, але для виконання дипломного проекту було використано express-validator. Validator.js загорнутий у Express-validator, який представляє його функціональність як серію проміжних програм.

Введені дані користувача передаються на сервер і перевіряються за допомогою однієї з серверних мов програмування, таких як JavaScript, TypeScript на платформі Node.js або Deno та інших у перевірці на стороні сервера. Після перевірки на стороні сервера зворотний зв’язок надається клієнту через нову динамічно створювану вебсторінку. Більш надійно перевіряти введення користувача на стороні сервера, оскільки це захищає від зловмисних користувачів, які можуть легко обійти скрипти на стороні клієнта та передати небезпечні дані на сервер.

3.7 Написання тестів для додатку

Тестування програмного забезпечення - це процес перевірки, чи відповідає фактичний програмний продукт очікуваним вимогам, і гарантує відсутність дефектів. Він включає в себе перевірку програмних та системних компонентів за допомогою ручних або автоматизованих методів, щоб оцінити одну або кілька властивостей, що цікавлять [33, 34].

По відношенню до реальних вимог мета тестування програмного забезпечення полягає в тому, щоб виявити помилки, прогалини або відсутні вимоги.

Jest - це платформа тестування з відкритим кодом на основі JavaScript, націлена насамперед на вебпрограми React. Коли модульні тести виконуються на інтерфейсі будь-якого програмного забезпечення, вони часто виявляються неефективними. Насамперед це пов’язано з тим, що модульні тести інтерфейсу вимагають складної та тривалої конфігурації. З фреймворком Jest цю складність можна значно мінімізувати.

На рисунку 3.11 зображено тестування клієнтської частини (Jest):

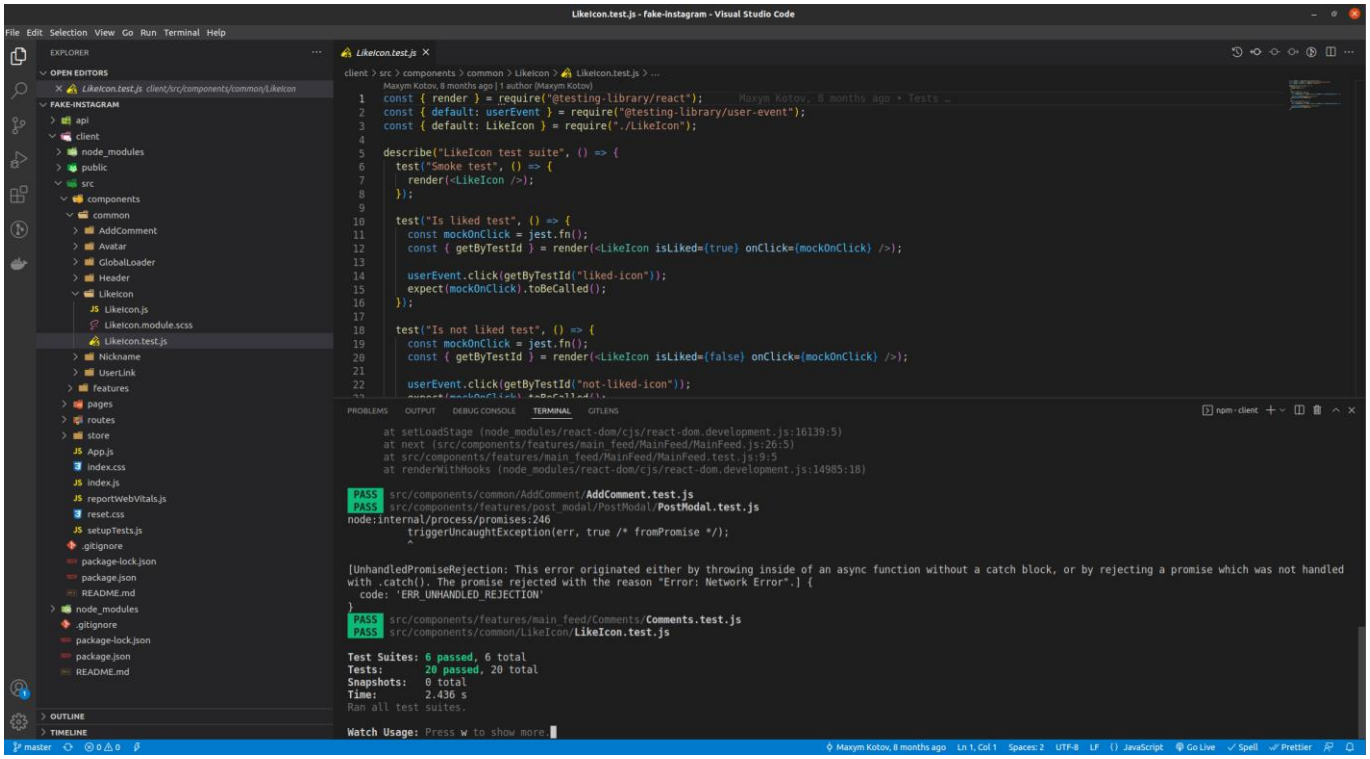

Рисунок 3.11 Тестування клієнтської частини (Jest)

Jest також можна використовувати для перевірки майже всього, що пов’язано з JavaScript, зокрема візуалізації вебпрограм у браузері. Jest також є популярним вибором для автоматизованого тестування браузера, що робить його однією з найбільш широко використовуваних платформ тестування Javascript [35].

Jest також поставляється зі змішаним пакетом, який включає бібліотеку тверджень, тестовий запуск і вбудовану бібліотеку підробки. Він виділяється своєю простотою, що робить його чудовим інструментом для тестування проектів бібліотеки JavaScript, таких як ReactJS, Node JS і TypeScript [35].

Тестування поведінки коду невеликими незалежними частинами називається модульним тестуванням. Одиниці часто призначені як найменша відповідна частина коду, який можна перевірити. На відміну від цього, інтеграційне тестування передбачає тестування групи модулів в цілому. Більшість розробників програмного забезпечення повинні вміти читати та писати модульні тести, але те, як проводяться інтеграційні тести, сильно відрізняється від продукту до продукту.

На рисунку 3.12 зображено тестування серверної частини (Chai & Mocha):

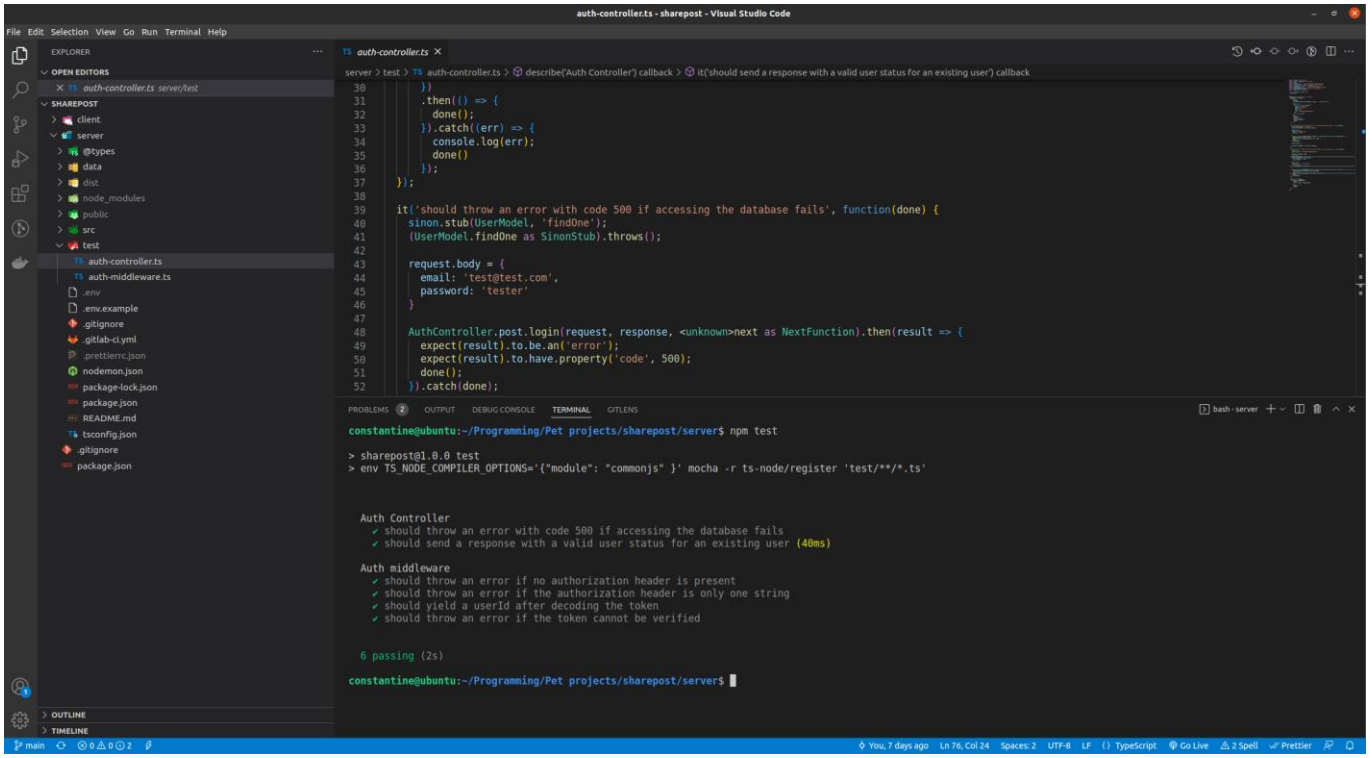

Рисунок 3.12 Тестування серверної частини (Chai & Mocha)

Фреймворки JavaScript Mocha і Chai часто використовуються разом для модульного тестування [36, 37].

Mocha - це платформа для тестування, яка надає функції, які запускаються в певному порядку та записують їх вихід у вікно терміналу [37].

Набір тестів - це група тестів, усі зосереджені на одній і тій же функції або поведінці. Тестовий приклад, часто відомий як модульний тест, є єдиним описом бажаної поведінки коду, який або проходить, або зазнає невдачі. Під ключовим словом describe набори тестів об’єднуються в пакети, а тестові приклади - за ключовим словом it [37].

Mocha також включає інструменти для очищення стану програмного забезпечення, що тестується, що гарантує, що тестові випадки виконуються незалежно один від одного. Інші інструменти можуть використовуватися, щоб заглушити або підробити необхідну поведінку інших блоків, з якими може взаємодіяти даний блок коду. Незалежність тестових випадків є важливим

принципом модульного тестування, оскільки він дозволяє точніше визначити причину збоїв у випадку невдачі тестового випадку, що прискорює процес налагодження.

Тестування програми має вирішальне значення, оскільки воно дозволяє виявити будь-які дефекти або помилки в програмному забезпеченні завчасно та виправити до того, як буде доставлений кінцевий продукт. Добре перевірене програмне рішення забезпечує надійність, безпеку та чудову продуктивність, що заощаджує час, гроші та підвищує рівень задоволеності клієнтів.

3.8 Формування практичних рекомендацій щодо розробки

Node.js - чудовий варіант для створення безпечного онлайн-додатка. Node.js використовується багатьма великими фірмами для веброзробки, тому що вона має сильну спільноту, чудові фреймворки та серверний JavaScript, який чудово підходить для додатків реального часу.

Однак Node.js уразливий до багатьох тих же ризиків, що й будь-яка інша платформа, як вже було розглянуто в цій роботі. Тож, потрібно дотримуватись наступних рекомендацій, аби розробляти безпечні додатки:

1. Валідація. потрібно валідувати дані користувачів для уникнення ін’єкцій.
2. Використання ODM та ORM. Оскільки сама по собі валідація не може уникнути всіх способів ін’єкцій, використання перевірених ODM та ORM дозволить нівелювати можливість таких ін’єкцій.
3. Використання токенів CSRF. Використання токенів в формах дозволить уникнути атаки CSRF.
4. Не потрібно повертати користувачу деталі помилки на сервері. Технічні деталі можуть видати зловмиснику залежності додатку, версії, тощо…
5. Потрібно уникати надсилання надмірної кількості даних на клієнт. Фільтрація даних повинна відбуватись на сервері.
6. Логування та моніторинг повинні бути налаштовані.

7. Потрібно уникати зберігання секретних ключів у файлах проекту. Для цього – варто використовувати змінні оточення.
8. Завжди не забувати використовувати безпечні HTTP заголовки.
9. Не запускати Node.js від імені привілейованих користувачів.
10. Тестувати додаток на відповідність очікуваній поведінці.
11. Слідкувати за пакетами від третіх сторін на виявлення в них вразливостей.
12. Використання TLS для безпечного обміну даними.

Висновки за розділом 3

Розробка вебпрограми полягає в тому, щоб встановити цілі та призначення програми. Інтерфейс користувача має бути розроблений з урахуванням відповіді на питання доцільності та ефективності використання такого додатка для вирішення прикладних та комерційних задач. Інформація про споживача буде надходити з користувацького інтерфейсу, тому розробники повинні створити програму такою, щоб вона отримувала та відповідала на цю інформацію.

Щоб забезпечити ефективне вирішення цих завдань веброзробкам, може бути корисно залучити надійних незалежних талантів - наприклад, розробників інтерфейсу. Це дає впевненість і розуміння, що даний вебдодаток створюють досвідчені професіонали.

Безпечна веброзробка - це ітеративний процес, який включає розробку, впровадження, тестування на вразливості та моніторинг. Відповідно до принципів безпечного кодування OWASP, дизайн програми повинен включати конфіденційність, цілісність і доступність, містити необхідні засоби контролю для запобігання несанкціонованої діяльності та забезпечувати поділ обов’язків. Після того, як програми написані, їх слід ретельно перевіряти на наявність вразливостей за допомогою комбінації інструментів сканування програм і перевірки коду.

ВИСНОВКИ

Безпека вебдодатків включає безліч методів і стратегій для захисту браузерів і програм. Ці стратегії захищають цифрові активи організації, такі як вебсайти, мобільні додатки, платіжні системи, тощо, від кіберзагроз. Більшість загроз безпеки вебдодатків виникають через наявні вразливості. Кіберзлочинці використовують сканери уразливостей вебсайтів, щоб використовувати слабкі місця та вразливості програм для крадіжки даних клієнтів для особистої вигоди.

Найпоширенішими цілями для атак на вебдодатки є системи керування вмістом, інструменти адміністрування баз даних і програми SaaS. Ці програми мають високу цінність і одна атака може завдати безпрецедентної шкоди, як-от:

1) втрата вихідного коду збільшує шанси на маніпулювання даними. Вихідні коди часто продаються на чорному ринку покупцям, які хочуть створювати високоякісні програми, але з меншим бюджетом;

2) втрата секретної інформації може створити такі проблеми, як вимоги викупу або крадіжки особистих даних;

3) неможливість забезпечення безпеки вебдодатків може призвести до проблем для ефективної роботи організацій. Без надійних заходів безпеки вони ризикують бути атакованими зловмисниками. Окрім фінансової та репутаційної шкоди, це також може призвести до судових позовів та стягнення звинувачень за дотримання вимог.

Тому запобігти атаці вебдодатків можливо лише за умов, що у команді розробників, персонал навчаний принципам розробки безпечних додатків, а у команді безпеки є досвідчений тестувальник коду та проникнення додатків. Також важливо отримати найновіші інструменти безпеки вебдодатків, щоб забезпечити безпеку від будь-яких інцидентів безпеки. Організації також повинні забезпечити певну практику, щоб уникнути порушень принципів розробки безпечних додатків.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. MONOLITH V/S Micro-Services Architecture. [Electronic resource]: Linkedin. – Access: https://www.linkedin.com/pulse/monolith-vs-micro-services-architecture-amodia-mca-pcep-pcap-csm/

2. O. Al-Debagy and P. Martinek, "A Comparative Review of Microservices and Monolithic Architectures," 2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI), 2018, pp. 000149-000154, doi: 10.1109/CINTI.2018.8928192.

3. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. [Electronic resource]: ResearchGate. – Access: https://www.researchgate.net/publication/304317852_Evaluating_the_monolithic_and_the_microservice_architecture_pattern_to_deploy_web_applications_in_the_cloud

4. Microservice architectures: more than the sum of their parts? [Electronic resource]: Digital Guide IONOS. – Access: https://www.ionos.com/digitalguide/websites/web-development/microservice-architecture/

5. N. Alshuqayran, N. Ali and R. Evans, "A Systematic Mapping Study in Microservice Architecture," 2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA), 2016, pp. 44-51, doi: 10.1109/SOCA.2016.15.

6. Pattern: Microservice Architecture. [Electronic resource]: Microservices.io. – Access: https://microservices.io/patterns/microservices.html

7. AWS Lambda Makes Serverless Applications A Reality. [Electronic resource]: Techcrunch. – Access: https://techcrunch.com/2015/11/24/aws-lamda-makes-serverless-applications-a-reality/

8. R. A. P. Rajan, "Serverless Architecture - A Revolution in Cloud Computing," 2018 Tenth International Conference on Advanced Computing (ICoAC), 2018, pp. 88-93, doi: 10.1109/ICoAC44903.2018.8939081.

9. B. Costa, P. F. Pires, F. C. Delicato and P. Merson, "Evaluating a Representational State Transfer (REST) Architecture: What is the Impact of REST in My

Architecture?," 2014 IEEE/IFIP Conference on Software Architecture, 2014, pp. 105-114, doi: 10.1109/WICSA.2014.29.

10. CHAPTER 5 Representational State Transfer (REST). [Electronic resource]: ICS. – Access: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

11. Mardan, A. (2018). Template Engines: Pug and Handlebars. In: Practical Node.js. Apress, Berkeley, CA. https://doi.org/10.1007/978-1-4842-3039-8_4

12. Template Engines: Jade and Handlebars. [Electronic resource]: ResearchGate. – Access: https://www.researchgate.net/publication/300170808_Template_Engines_Jade_and_Handlebars

13. Olaf Hartig and Jorge Pérez. 2018. Semantics and Complexity of GraphQL. In Proceedings of the 2018 World Wide Web Conference (WWW). International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 1155–1164. https://doi.org/10.1145/3178876.3186014

14. Research of Web Real-Time Communication Based on Web Socket. [Electronic resource]: ResearchGate. – Access: https://www.researchgate.net/publication/276491172_Research_of_Web_Real-Time_Communication_Based_on_Web_Socket

15. Escalation of Privilege. [Electronic resource]: ResearchGate. – Access: https://www.researchgate.net/publication/343616729_Escalation_of_Privilege

16. HTML Code Injection and Cross-site Scripting. [Electronic resource]: ResearchGate. – Access: https://www.researchgate.net/publication/293454449_HTML_Code_Injection_and_Cross-site_Scripting

17. B. Hou, K. Qian, L. Li, Y. Shi, L. Tao and J. Liu, "MongoDB NoSQL Injection Analysis and Detection," 2016 IEEE 3rd International Conference on Cyber Security and Cloud Computing (CSCloud), 2016, pp. 75-78, doi: 10.1109/CSCloud.2016.57.

18. Cryptography for Network Security: Failures, Successes and Challenges. [Electronic resource]: ResearchGate. – Access:

https://www.researchgate.net/publication/225155362_Cryptography_for_Network_Security_Failures_Successes_and_Challenges

19. J. H. Moore, "Protocol failures in cryptosystems," in Proceedings of the IEEE, vol. 76, no. 5, pp. 594-602, May 1988, doi: 10.1109/5.4444.

20. Automatic Detection of Security Misconfigurations in Web Applications. [Electronic resource]: ResearchGate. – Access: https://www.researchgate.net/publication/347196496_Automatic_Detection_of_Security_Misconfigurations_in_Web_Applications

21. B. Eshete, A. Villafiorita and K. Weldemariam, "Early Detection of Security Misconfiguration Vulnerabilities in Web Applications," 2011 Sixth International Conference on Availability, Reliability and Security, 2011, pp. 169-174, doi: 10.1109/ARES.2011.31.

22. Cross-site request forgery. [Electronic resource]: Shiflett. – Access: https://shiflett.org/articles/cross-site-request-forgeries

23. X. Lin, P. Zavarsky, R. Ruhl and D. Lindskog, "Threat Modeling for CSRF Attacks," 2009 International Conference on Computational Science and Engineering, 2009, pp. 486-491, doi: 10.1109/CSE.2009.372.

24. S. Tilkov and S. Vinoski, "Node.js: Using JavaScript to Build High-Performance Network Programs," in IEEE Internet Computing, vol. 14, no. 6, pp. 80-83, Nov.-Dec. 2010, doi: 10.1109/MIC.2010.145.

25. Node.js Challenges in Implementation. [Electronic resource]: ResearchGate. – Access: https://www.researchgate.net/publication/318310544_Nodejs_Challenges_in_Implementation

26. K. Saundariya, M. Abirami, K. R. Senthil, D. Prabakaran, B. Srimathi and G. Nagarajan, "Webapp Service for Booking Handyman Using Mongodb, Express JS, React JS, Node JS," 2021 3rd International Conference on Signal Processing and Communication (ICPSC), 2021, pp. 180-183, doi: 10.1109/ICSPC51351.2021.9451783.

27. How to use sessions to identify users across requests. [Electronic resource]: Flaviocopes: https://flaviocopes.com/express-sessions/

28. JWT Handbook. [Electronic resource]: Auth0. – Access: https://auth0.com/resources/ebooks/jwt-handbook

29. HTTP SECURITY HEADERS. [Electronic resource]: ResearchGate. – Access: https://www.researchgate.net/publication/354373317_HTTP_SECURITY_HEADERS

30. A. Lavrenovs and F. J. R. Melón, "HTTP security headers analysis of top one million websites," 2018 10th International Conference on Cyber Conflict (CyCon), 2018, pp. 345-370, doi: 10.23919/CYCON.2018.8405025.

31. Node.js CSRF Protection Guide: Examples and How to Enable It. [Electronic resource]: Stackhawk. – Access: https://www.stackhawk.com/blog/node-js-csrf-protection-guide-examples-and-how-to-enable-it/

32. Data Validation. [Electronic resource]: ResearchGate. – Access: https://www.researchgate.net/publication/347625018_Data_Validation

33. S. M. Srinivasan and R. S. Sangwan, "Web App Security: A Comparison and Categorization of Testing Frameworks," in IEEE Software, vol. 34, no. 1, pp. 99-102, Jan.-Feb. 2017, doi: 10.1109/MS.2017.21.

34. Contract Driven Development = Test Driven Development – Writing Test Cases. [Electronic resource]: Leitner, Andreas; Ciupa, Ilinca; Oriol, Manuel; Meyer, Bertrand; Fiva, Arno (September 2007). – Access: http://se.inf.ethz.ch/people/leitner/publications/cdd_leitner_esec_fse_2007.pdf

35. Basic Testing with Jest. [Electronic resource]: ResearchGate. – Access: https://www.researchgate.net/publication/333291287_Basic_Testing_with_Jest

36. Chai Assertion Library. [Electronic resource]: Chai. – Access: https://www.chaijs.com/

37. Mardan, A. (2018). TDD and BDD for Node.js with Mocha. In: Practical Node.js. Apress, Berkeley, CA. https://doi.org/10.1007/978-1-4842-3039-8_3

ДОДАТКИ

ДОДАТОК А
ЛІСТИНГ КОДУ ДОДАТКУ ENROLLAPP

Middleware аутентифікації:

```
const CourseService = require("../services/CourseService");

function checkAuth(req, res, next) {
  if (!req.session.isAuthenticated) {
    return res.redirect(303, '/api/auth/login');
  } else {
    next();
  }
}

async function checkAuthor(req, res, next) {
  const courseId = req.params.id;
  const userId = req.session.user?._id;

  const isAuthor = await CourseService.checkAuthor(courseId, userId);

  if (!isAuthor) {
    return res.redirect(303, '/api/auth/login');
  } else {
    next();
  }
}

function checkAccountOwner(req, res, next) {
  const submitterId = req.params.id;
  const userId = req.session.user?._id;

  const isOwner = submitterId === userId;

  if (!isOwner) {
    return res.redirect(303, '/api/auth/login');
  } else {
    next();
  }
}

module.exports = {
  checkAuth,
  checkAuthor,
  checkAccountOwner
}
```

Котролер POST запитів аутентифікації:

```
const UserService = require("../../../services/UserService");

async function createUser(req, res) {
  const body = req.body;
  try {
    await UserService.createUser(body);
  } catch (createError) {
    if ( createError.code === 11000 ) {
      return res.status(400).json({ message: 'User already exists', duplicate: true
});
    }

    return res.status(500).json({ message: 'Error while saving user'});
  }

  res.sendStatus(201);
}

async function authUser(req, res) {
  const email = req.body.email;
  const password = req.body.password;
  let userData;

  try {
    userData = await UserService.authUser(email, password);
  } catch (authError) {
    if (authError.httpStatus) {
      return res.status(401).json({ message: authError.message });
    }
    return res.status(500).json({ message: 'Error while authenticating user' });
  }

  req.session.isAuthenticated = userData.isAuthenticated;
  req.session.user = userData.userInfo;

  res.redirect(303, '/api/main/courses');
}

async function confirmReset(req, res) {
  const email = req.body.email;
  let userId;
  try {
    userId = await UserService.generateResetToken(email);
  } catch (resetError) {
    return res.status(400).json({ message: resetError.message });
  }

  res.json({ userId });
}

const postController = {
  createUser,
  authUser,
  confirmReset
};

module.exports = postController;
```

Котролер PATCH запитів аутентифікації:

```
const UserService = require("../../../services/UserService");

async function changePassword(req, res) {
  const { token, userId, newPassword } = req.body;

  try {
    await UserService.resetPassword(token, userId, newPassword);
  } catch (resetError) {
    return res.status(400).json({ message: resetError.message });
  }

  return res.redirect(303, '/api/auth/login');
}

const patchController = {
  changePassword
};

module.exports = patchController;
```

Котролер DELETE запитів аутентифікації:

```
function logout(req, res) {
  req.session.destroy(function(err) {
    if (err) {
      res.sendStatus(400);
    } else {
      res.redirect(303, '/api/main/courses');
    }
  });
}

const deleteController = {
  logout
};

module.exports = deleteController;
```

ДОДАТОК Б
ЛІСТИНГ КОДУ ДОДАТКУ KO-SHOP

Middleware аутентифікації:

```
import { NextFunction, Request, Response } from 'express';

function isLoggedIn(req: Request, res: Response, next: NextFunction) {
  if (!req.session.isLoggedIn) {
    return res.redirect('/api/auth/login');
  }

  next();
}

export const AuthMiddleware = {
  isLoggedIn
}
```

Котролер POST запитів аутентифікації:

```
import { NextFunction, Request, Response } from 'express';
import bcrypt from 'bcryptjs';
import transporter from '../../../config/email';
import { BASE_LINK, EMAIL_SENDER } from '../../../config';
import crypto from 'crypto';
import { UserInterface } from '../../../interfaces/User.interface';
import { validationResult } from 'express-validator';
import UserService from '../../../services/UserService';

function login(req: Request, res: Response, next: NextFunction) {
  const email = req.body.email;
  const password = req.body.password;
  const errors = validationResult(req);

  if (!errors.isEmpty()) {
    return res.status(422).render('auth/login', {
      path: '/api/auth/login',
      pageTitle: 'Login',
      errorMessage: errors.array()[0].msg,
      oldInput: {
        email: email,
        password: password
      },
      validationErrors: errors.array()
    });
  }

  UserService.findUser({ email })
    .then(user => {
      if (!user) {
        req.flash('error', 'Invalid email or password.');
        return req.session.save(() => res.redirect('/api/auth/login'));
      }
```

```
      bcrypt
        .compare(password, user.password)
        .then(doMatch => {
          if (doMatch) {
            req.session.isLoggedIn = true;
            req.session.user = user;

            return req.session.save(err => {
              console.log(err);
              res.redirect('/');
            });
          }

          req.flash('error', 'Invalid email or password.');
          req.session.save(() => res.redirect('/api/auth/login'));
        })
        .catch(err => {
          console.log(err);
          res.redirect('/api/auth/login');
        });
    })
    .catch(err => {
      console.log(err);
      next(err);
    });
};

function signup(req: Request, res: Response, next: NextFunction) {
  const email: string = req.body.email;
  const password: string = req.body.password;
  const confirmPassword: string = req.body.confirmPassword;
  const errors = validationResult(req);

  if (!errors.isEmpty()) {
    console.log(errors.array());
    return res.status(422).render('auth/signup', {
      path: '/api/auth/signup',
      pageTitle: 'Signup',
      errorMessage: errors.array()[0].msg,
      oldInput: {
        email: email,
        password: password,
        confirmPassword: confirmPassword
      },
      validationErrors: errors.array()
    });
  }

  UserService.findUser({ email })
    .then(userDoc => {
      if (userDoc) {
        req.flash('error', 'E-Mail exists already, please pick a different one.');
        req.session.save(() => res.redirect('/api/auth/signup'));
        return
      }

      return bcrypt
        .hash(password, 12)
        .then(hashedPassword => {
          return UserService.createUser({
            email: email,
            password: hashedPassword,
            cart: { items: [] }
          });
```

```
        })
        .then(() => {
          res.redirect('/api/auth/login');

          transporter.sendMail({
            to: email,
            from: EMAIL_SENDER,
            subject: 'Signup succeeded!',
            html: '<h1>You successfully signed up!</h1>'
          });
        });
    })
    .catch(err => {
      console.log(err);
      next(err);
    });
};

function logout(req: Request, res: Response, next: NextFunction) {
  req.session.destroy(err => {
    if (err) {
      console.log(err);
      next(err);
    }

    res.redirect('/');
  });
};

function reset(req: Request, res: Response, next: NextFunction) {
  crypto.randomBytes(32, (err, buffer) => {
    if (err) {
      console.log(err);
      return res.redirect('/reset');
    }

    const token = buffer.toString('hex');

    UserService.findUser({ email: req.body.email })
      .then(user => {
        if (!user) {
          req.flash('error', 'No account with that email found.');
          req.session.save(() => res.redirect('/api/auth/reset'));
          return
        }

        user.resetToken = token;
        user.resetTokenExpiration = Date.now() + 3600000;

        return user.save();
      })
      .then(() => {
        res.redirect('/');

        return transporter.sendMail({
          to: req.body.email,
          from: EMAIL_SENDER,
          subject: 'Password reset',
          html: `
            <p>You requested a password reset</p>
            <p>Click this <a href="${BASE_LINK}/api/auth/reset/${token}">link</a> to
set a new password.</p>
          `
        });
```

```
      })
      .catch(err => {
        console.log(err);
        next(err);
      });
  });
};

function newPassword(req: Request, res: Response, next: NextFunction) {
  const newPassword = req.body.password;
  const userId = req.body.userId;
  const passwordToken = req.body.passwordToken;
  let resetUser: UserInterface;

  UserService.findUser({
    resetToken: passwordToken,
    resetTokenExpiration: { $gt: Date.now() },
    _id: userId
  })
    .then(user => {
      if (!user) {
        req.flash('error', 'Refresh token not found');
        req.session.save(() => res.redirect('/api/auth/login'));
        return
      }

      resetUser = user;
      return bcrypt.hash(newPassword, 12);
    })
    .then(hashedPassword => {
      resetUser.password = hashedPassword!;
      resetUser.resetToken = undefined;
      resetUser.resetTokenExpiration = undefined;

      return resetUser.save();
    })
    .then(() => {
      res.redirect('/api/auth/login');
    })
    .catch(err => {
      console.log(err);
      next(err);
    });
};

export const postController = {
  login,
  signup,
  logout,
  reset,
  newPassword
}
```

ДОДАТОК В
ЛІСТИНГ КОДУ ДОДАТКУ SHAREPOST

Middleware аутентифікації:

```
import { NextFunction, Request, Response } from 'express';
import HTTPError from '../util/errors/baseHTTP.error';
import jwt, { JwtPayload } from 'jsonwebtoken';
import { JWT_SECRET } from '../config';

function isLoggedIn(req: Request, res: Response, next: NextFunction) {
  const authHeader = req.get('Authorization');

  if (!authHeader) {
    return next(new HTTPError({ message: 'Not authenticated.', code: 401 }));
  }

  const token = authHeader.split(' ')[1];
  let decodedToken;

  try {
    decodedToken = jwt.verify(token, JWT_SECRET);
  } catch (err) {
    return next(new HTTPError({ message: 'Token compromised.', code: 401 }));
  }

  if (!decodedToken) {
    return next(new HTTPError({ message: 'Not authenticated.', code: 401 }));
  }

  req.userId = (decodedToken as JwtPayload).userId;
  next();
}

export {
  isLoggedIn
}
```

Котролер POST запитів аутентифікації:

```
import { NextFunction, Request, Response } from "express";
import { UserInterface } from "../../../interfaces/User.interface";
import { UserModel } from "../../../models/User.model";
import HTTPError from "../../../util/errors/baseHTTP.error";
import bcrypt from 'bcryptjs';
import jwt from 'jsonwebtoken';
import { JWT_SECRET } from "../../../config";

async function login(req: Request, res: Response, next: NextFunction) {
  const email = req.body.email;
  const password = req.body.password;
  let loadedUser: UserInterface;

  try {
    const user = await UserModel.findOne({ email: email });

    if (!user) {
      throw new HTTPError({ message: 'Authentication failure.', code: 401 });
    }

    loadedUser = user;
    const isEqual = await bcrypt.compare(password, user.password);

    if (!isEqual) {
      throw new HTTPError({ message: 'Authentication failure.', code: 401 });
    }

    const token = jwt.sign(
      {
        email: loadedUser.email,
        userId: loadedUser._id.toString()
      },
      JWT_SECRET,
      { expiresIn: '6h' }
    );

    res.status(200).json({ token: token, userId: loadedUser._id.toString() });
    return;
  } catch (err) {
    if (err instanceof HTTPError) {
      next(err);
      return err;
    } else if (err instanceof Error) {
      const httpError = new HTTPError({ message: err.message });

      next(httpError);
      return httpError;
    }
  }
};

export const postController = {
  login
}
```

Котролер PATCH запитів аутентифікації:

```
import { NextFunction, Request, Response } from "express";
import { UserModel } from "../../../models/User.model";
import HTTPError from "../../../util/errors/baseHTTP.error";

async function updateUserStatus(req: Request, res: Response, next: NextFunction) {
  const newStatus = req.body.status;

  UserModel.findById(req.userId)
    .then(user => {
      if (!user) {
        throw new HTTPError({ message: 'User not found.', code: 404 });
      }

      user.status = newStatus;
      return user.save();
    })
    .then(() => {
      res.status(200).json({ message: 'User updated.' });
    })
    .catch(err => {
      if (!(err instanceof Error)) return next(err);
      const httpError = new HTTPError({ message: err.message });

      next(httpError);
    });
};

export const patchController = {
  updateUserStatus
}
```

Котролер PUT запитів аутентифікації:

```
import { NextFunction, Request, Response } from "express";
import { validationResult } from "express-validator";
import { UserModel } from "../../../models/User.model";
import HTTPError from "../../../util/errors/baseHTTP.error";
import bcrypt from 'bcryptjs';

async function signup(req: Request, res: Response, next: NextFunction) {
  const errors = validationResult(req);

  if (!errors.isEmpty()) {
    return next(new HTTPError({ message: 'Validation failed, entered data is
incorrect.', code: 422 }));
  }

  const email = req.body.email;
  const name = req.body.name;
  const password = req.body.password;

  bcrypt
    .hash(password, 12)
    .then(hashedPw => {
      const user = new UserModel({
        email: email,
        password: hashedPw,
        name: name
      });

      return user.save();
    })
    .then(result => {
      res.status(201).json({ message: 'User created!', userId: result._id });
    })
    .catch(err => {
      if (!(err instanceof Error)) return next(err);
      const httpError = new HTTPError({ message: err.message });

      next(httpError);
    });
};

export const putController = {
  signup
}
```

Тести для контролера аутентифікації:

```
import { expect } from 'chai';
import sinon, { SinonStub } from 'sinon';
import 'mocha';
import HTTPError from '../src/util/errors/baseHTTP.error';
import { NextFunction, request, response } from 'express';
import { UserModel } from '../src/models/User.model';
import AuthController from '../src/api/controllers/auth/index';
import { DB_CONNECTION_STRING } from '../src/config';
import mongoose from 'mongoose';

function next(err?: HTTPError) {
  return err;
}

describe('Auth Controller', function() {
  before(function(done) {
    mongoose
      .connect(
        DB_CONNECTION_STRING.replace('sharepost', 'sharepost-test')
      )
      .then(() => {
        const user = new UserModel({
          email: 'test@test.com',
          password: 'tester',
          name: 'Test',
          posts: [],
          _id: '5c0f66b979af55031b34728a'
        });
        return user.save();
      })
      .then(() => {
        done();
      }).catch((err) => {
        console.log(err);
        done()
      });
  });

  it('should throw an error with code 500 if accessing the database fails',
function(done) {
    sinon.stub(UserModel, 'findOne');
    (UserModel.findOne as SinonStub).throws();

    request.body = {
      email: 'test@test.com',
      password: 'tester'
    }

    AuthController.post.login(request, response, <unknown>next as
NextFunction).then(result => {
      expect(result).to.be.an('error');
      expect(result).to.have.property('code', 500);
      done();
    }).catch(done);

    request.body = {};

    (UserModel.findOne as SinonStub).restore();
  });
```

```
  it('should send a response with a valid user status for an existing user',
function(done) {
    //@ts-ignore
    request.userId = '5c0f66b979af55031b34728a';

    response.statusCode = 500;
    //@ts-ignore
    response.userStatus = null;
    response.status = function(code) {
      this.statusCode = code;
      return this;
    }
    //@ts-ignore
    response.json =function(data) {
      //@ts-ignore
      this.userStatus = data.status;
    }

    AuthController.get.userStatus(request, response, <unknown>next as
NextFunction).then(() => {
      expect(response.statusCode).to.be.equal(200);
      //@ts-ignore
      expect(response.userStatus).to.be.equal('I am new!');
      done();
    }).catch(done);
  });

  after(function(done) {
    UserModel.deleteMany({})
      .then(() => {
        return mongoose.disconnect();
      })
      .then(() => {
        done();
      });
  });
});
```

Тести для middleware аутентифікації:

```
import { expect } from 'chai';
import jwt from 'jsonwebtoken';
import sinon from 'sinon';
import 'mocha';
import { isLoggedIn } from '../src/middleware/auth';
import HTTPError from '../src/util/errors/baseHTTP.error';
import { NextFunction, request, response } from 'express';

function next(err?: HTTPError) {
  if (!err) return;
  throw new Error(err.message);
}

describe('Auth middleware', function() {
  beforeEach(() => {
    sinon.stub(request, 'get');
  });

  afterEach(() => {
    (request.get as sinon.SinonStub).restore();
  });

  it('should throw an error if no authorization header is present', function() {
    (request.get as sinon.SinonStub).returns(null);
    expect(isLoggedIn.bind(this, request, response, next as NextFunction)).to.throw(
      'Not authenticated.'
    );
  });

  it('should throw an error if the authorization header is only one string',
function() {
    (request.get as sinon.SinonStub).returns('null');
    expect(isLoggedIn.bind(this, request, response, next as
NextFunction)).to.throw();
  });

  it('should yield a userId after decoding the token', function() {
    (request.get as sinon.SinonStub).returns('Bearer djfkalsdjfaslfjdlas');
    sinon.stub(jwt, 'verify');
        (jwt.verify as sinon.SinonStub).returns({ userId: 'abc' });
    isLoggedIn.call(this, request, response, next as NextFunction);
    expect(request).to.have.property('userId');
    expect(request).to.have.property('userId', 'abc');
    expect((jwt.verify as sinon.SinonStub).called).to.be.true;

    (jwt.verify as sinon.SinonStub).restore();
  });

  it('should throw an error if the token cannot be verified', function() {
    (request.get as sinon.SinonStub).returns('Bearer xyz');
    expect(isLoggedIn.bind(this, request, response, next as
NextFunction)).to.throw();
  });
});
```

Налаштування CORS:

```
import { NextFunction, Request, Response } from 'express';

function configCors(req: Request, res: Response, next: NextFunction) {
  res.setHeader('Access-Control-Allow-Origin', '*');
  res.setHeader(
    'Access-Control-Allow-Methods',
    'OPTIONS, GET, POST, PUT, PATCH, DELETE'
  );
  res.setHeader('Access-Control-Allow-Headers', 'Content-Type, Authorization');

  next();
}

export default configCors;
```

ДОДАТОК Г
СПИСОК ОПУБЛІКОВАНИХ ПРАЦЬ ЗА ТЕМОЮ ДИПЛОМНОЇ РОБОТИ

Статті у наукових фахових виданнях України

Стаття індексована у Scopus: "RF signals encryption with AES in WDID Toliupa, S., Nakonechnyi, V., Kotov, M., Solodovnyk, V. CEUR Workshop Proceedingsthis link is disabled, 2021, 2845, pp. 96–105".

Тези наукових доповідей

1. Тези у "VII Міжнародній науково-технічній конференції “Захист інформації і безпека інформаційних систем” (30-31.05. 2019 р., Україна, Львів)". Тема: "Kotov M.S., Serhii Toliupa. Intrusion Prevention System (IPS)".

2. Тези у "IT&I 2019 Information Technology and Interactions". Тема: "Toliupa S., Nakonechnyi V., Druzhinin V., Kotov M. Model of the process of optimal planning of the modular structure of an information security system".

3. Тези у "IT&I 2020 Information Technology and Interactions". Тема: "RF Signals Encryption with AES in WDID96-105 Serhii Toliupa, Volodymyr Nakonechnyi, Maksym Kotov, Valeriia Solodovnyk".

4. Тези у "МІЖНАРОДНА НАУКОВО-ПРАКТИЧНА КОНФЕРЕНЦІЯ «ПРИКЛАДНІ СИСТЕМИ ТА ТЕХНОЛОГІЇ В ІНФОРМАЦІЙНОМУ СУСПІЛЬСТВІ»". Тема: "Fesenko A., Ponomarenko Y., Kotov M., Rudenko K. Data transfer technologies between password managers".

5. Тези та доповідь у "2019 II Міжнародна науково-практична конференція “Проблеми кібербезпеки інформаційно-телекомунікаційних систем” (PCSITS)". Тема: "Kotov M., Toliupa S., Bohuslavska O. SYMMETRIC ALGORITHM OF BLOCK ENCRYPTION OR ADVANCED ENCRYPTION STANDARD".

6. Тези та доповідь у "2020 III Міжнародна науково-практична конференція “Проблеми кібербезпеки інформаційно-телекомунікаційних систем” (PCSITS)".

Тема: "Asymmetric cryptographic algorithm or public key cryptographic algorithm RSA.".

7. Тези у "2020 III Міжнародна науково-практична конференція “Проблеми кібербезпеки інформаційно-телекомунікаційних систем” (PCSITS)". Тема: "ANALYSIS OF NETWORK STEGANOGRAPHY METHODS".

8. Тези, доповідь, сертифікат та участь в організації (секретар) у "2021 IV Міжнародна науково-практична конференція “Проблеми кібербезпеки інформаційно-телекомунікаційних систем” (PCSITS)". Тема: "Analysis of reliability, security, and vulnerabilities of the Transport Layer Security protocol Serhii Toliupa, Volodymyr Nakonechnyi, Maksym Kotov".